Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1997

Performance Analysis and Visualization Tools for Parallel Computing

K. N. Pantazopoulos

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

Report Number:
97-006

Pantazopoulos, K. N. and Houstis, Elias N., "Performance Analysis and Visualization Tools for Parallel Computing" (1997). *Department of Computer Science Technical Reports.* Paper 1346.
https://docs.lib.purdue.edu/cstech/1346

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

PERFORMANCE ANALYSIS AND VISUALIZATION TOOLS FOR PARALLEL COMPUTING

K. N. Pantazopoulos
E. N. Houstis

Departmemt of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR 97-006
January 1997

Performance Analysis and Visualization Tools for Parallel Computing

K.N.Pantazopoulos

E.N. Houstis

In this report we present the results of our work on the Performance Analysis and Visualization Tools for Parallel machines. We discuss from various perspectives the need for such tools, we present a survey of existing tools and we evaluate them based on a set of qualitative and quantitative characteristics. We continue with a critical discussion of the state-of-the-art and the challenges that exist and need to be faced. We conclude with a presentation and analysis of our view for a competitive performance analysis and visualization tool and we document briefly its design and prototyping status.

January 21, 1997

Table of Contents

List of Figures

List of Tables

Introduction

The need to analyse the performance of computer systems is well understood and documented. It is a direct way to assess the functionality and operation of a system against the goals set out for its use. Substantial efforts over the years have contributed to the successful establishment of both analytic and experimental frameworks for measuring and analysing the performance of sequential computer systems. However, the challenge of performance analysis and evaluation for parallel systems is by far harder and more demanding. The inherent complexity of parallel systems as well as the lack of standards and uniformity has prevented the development of widely available and applicable techniques. Nevertheless, considerable progress has been made towards this goal and a number of valuable contributions are today available for performance analysis and visualization of parallel systems.

Parallel systems come in a variety of shapes and features. There are a number of different architectures of parallel machines. To mention a popular classification, there are vector computers, shared memory computers and distributed memory computers. To make things more complicated, there are also hybrid computers that adopt more that one architectures. For example one can get a clustered distributed memory computer, which provides shared memory features on each of its clusters and where the individual processing elements of each cluster have vector processing capabilities.

The heterogeneity in architectures is accompanied by the heterogeneity in programming and development environments. Data parallel programming, message passing programming, and a number of hybrid systems complete the picture of the available parallel computing infrastructure.

At the same time, the diversity of applications that are available on parallel machines adds even further complexity to the parallel systems puzzle. Scientific applications that are mainly number crunching, transaction oriented applications that are mainly I/O oriented, and real time applications that are mainly mission oriented are all types of applications commonly used on parallel computers. Needless to say that each family of applications has different requirements and consequently there are different characteristics of parallel systems infrastructure that affect its performance.

In order to built widely acceptable and applicable tools one would have to account for all the various combinations and provide an abstraction that would govern the behaviour of the various architecture/application pairs. Unfortunately this seems hopeless and until today no such general methodology exists. Instead practitioners and scientists have to come up with approximate solutions to the performance analysis and visualization problem, often customized to the scenario at hand. This is the case for many performance analysis and visualization systems that come with a specific parallel machine and are customized to analyse the behaviour of that machine only but also for many systems that customized for a specific programming model.

In this report we focus on systems and techniques that are mainly applicable to distributed memory machines that have a message passing programming environment. This

includes systems where the programming environment might be data parallel, such as HPF, but which are effectively implemented through message passing. The reasons we select to deal with these systems are several. Distributed memory systems are the most recent member of the parallel systems family and thus less analysed. They provide the necessary mechanisms to scale up to thousands of processors (MPPs) providing enormous computing power. They pose problems that are not faced in shared memory or vector systems. For example instrumentation techniques are much more involved and visualization extremely complicated.

In the first part of this report we survey some of the most popular tools and the nomenclature that governs the field.

In the second part we present our approach in designing a system. More details of its design can be found in the appendix.

Survey

In this part we discuss the issue of performance analysis and visualization from various perspectives. First we go through a brief introduction of the main concepts, terminology and jargon that is encountered in the area. We then discuss some of the most important tools along with various other findings that are influencing further development.

1.0 Main Concepts and Terminology

Generally speaking one can divide the performance analysis and visualization systems in three components. The first is the **monitoring** or **instrumentation** component. This system is responsible for collecting information from the parallel machine, the programming environment and the application that will later be used to analyse the performance at the various levels. The second is the **trace data** component. This component is where the information from monitoring is stored. The third component is the **analysis** and **visualization** component which will perform the actual task using the information stored in the traced data. Sometimes we refer to the monitoring component as the back end of the system and the analysis and visualization as the front end. We do not explicitly mention the trace data since they are assumed to be available. What provides the conceptual link between the front end and the back end is the set of measurements and analysis algorithms that one wants to apply to the parallel system. This set is the guideline that defines what should the trace data component contain and thus effectively what should the monitoring component provide. Furthermore the set of measurements and analysis algorithms prescribes what should the front end present.

With the above abstraction in mind we proceed to describe the various types of measurements, instrumentation and analysis appoaches that one could follow to structure a complete performance analysis and visualization system. A number of researchers discuss the issue [6,30,39,46,47,48].

1.1 Types of Measurements

Most performance measures treat one or more of the following issues:

- how quickly a given task can be accomplished
- how well a system can deal with failures and other "unusual" situations and,
- how effectively the system uses the available resources.

Based on the above we can categorize the performance measures of interest as follows:

- **Responsiveness:** it indicates how fast the system under measurement can accomplish a given task. Possible measures related to responsiveness include *waiting time*, *processing time*, *queue length* e.t.c.
- **Usage Level:** evaluates the degree of utilization of the various components of a system. Examples of this measure are the *throughput* and the *usage*. The objectives

behind these measures are often in conflict with those of the responsiveness, since a well utilized system will in general respond slower.

- **Missionability**: indicates whether the system will remain operational for the duration of a task. Possible measures include the *distribution of work*, the *interval availability* and the *lifetime*.
- **Dependability**: indicates how reliable the system is in the long run. This category includes measures such as *mean time to failure*, *long-term availability*, *cost of failure*.
- **Productivity**: evaluates how effectively a user can accomplish a task. Examples include measures such as *user friendliness*, and *maintainability*.
- **Performability**: this category is often applied to contemporary distributed and parallel systems, where several operational levels are possible. Performability measures include:
  1. instantaneous availability: the probability that the measured system is operational at a given time.
  2. interval availability: the fraction of time during a given time interval, during which the measured system was operational.
  3. cumulative performance: the total work accomplished during a time interval.
  4. computational capacity: average work rate during a time interval.

Depending on the application that is being analyzed, the relative importance of the above measure categories can vary. A typical diversification is given as:

- **general purpose computing**: relevant measures are repsonsiveness, usage level and productivity.
- **high availability computing**: (includes transaction processing computing). relevant measures include responsiveness, dependability and productivity.
- **real time computing**: relevant measures include responsiveness and dependability. Utilization and throughput play little role in this area.
- **mission oriented**: relevant measures include reliability over a short period and responsiveness.
- **long-life computing**: relevant measures include dependability and responsiveness.

There are three basic techniques for performance evaluation:

- **measurement,**
- **simulation,**
- **analytic modeling**

The suitability of one technique or another depends on several factors, such as the stage of the design of the application to be evaluated or the availability of performance data. As an example, if we want to evaluate the performance of a given feature of an application, that is still under development, measurements are not possible, whereas the choice between simulation and analytic modeling depends on the complexity and the accuracy of the evaluation.

The general classes of measurements identified above are the same for conventional as well as for parallel systems. What is actually different is the way the measures are carried out, and the way the high level description of a performance analysis measure translates into implementation techniques. The feature that mainly characterizes parallel systems with respect to conventional ones is the presence of more than one measurement context. In general we can identify at least three such contexts:

- the **local** context which refers to what happens with a single processing element (e.g. a process, a processor) of the parallel system,
- the **interprocess/interprocessor** context which refers to what happens between cooperating processing elements, and,
- the **global** context which refers to what happens to the parallel system as a whole.

Measurements performed at one level can provide information for the other levels. For example, the communication between processes residing on different processors contribute to the definition of the interconnection medium load at a global level. Conversely, the communication latency between two processes depends also on the global system conditions.

1.2 Types of Instrumentation

Instrumentation environments can be classified according to different criteria, that are often related to the context where they are to be used. One broad classification refers to the way the instrumentation environments are implemented. Taking this perspective we have:

- **hardware** monitors, consisting of specialized hardware plugged on the target system,
- **software** monitors, consisting entirely of software modules embedded in the system and/or application software of the target system, and,
- **hybrid** monitors, consisting of a mix of hardware and software modules cooperating.

Another common classification refers to the way the instrumentation collects the information. Taking this perspective we have:

- **sampled** instrumentation, and,
- **traced** instrumentation.

In sampled instrumentation, the monitor takes samples of some system or application resources and checks their state. A typical example of sampled instrumentation occurs with time driven monitors, consisting of one or more tasks that periodically wake up and check the system state. Sampled instrumentation is simpler to implement than traced instrumentation, but it provides only statistical information about the target system, and often does not meet the performance analysis objectives.

In traced instrumentation, the monitor actions are consequences of events that bring the system or the application to specified state. An example of traced instrumentation occurs during debugging, where the presence of breakpoints in the program triggers the

debugger. Traced instrumentation is the one most often used, as it allows for richer measurement options. Within the traced instrumentation category we can further subdivide into three categories:

- **event detection**: gives information only about the occurrence of an event,
- **event trace**: gives information about the occurrence of an event and how this event was produced,
- **full trace instrumentation**: gives full information about the events produced and their origins.

Finally, another aspect that can be taken into account to classify instrumentation environments refers to the way they interact with the target system. In this case we can have:

- **intrusive** instrumentation, and,
- **nonintrusive** instrumentation.

1.3 Types of Performance Analysis and Visualization

In the real world the performance analysis and visualization systems usually come together with an instrumentation system that will provide the data to be analysed. Only recent years it has become evident to the parallel computing community that many advantages can be gained by separating the monitoring and instrumentation part from the analysis and visualization part of the systems.

Analysis and Visualization can be classified according to the way the data are fed in the system. Moreover, each category implies a specific type of instrumentation and degree of intrusion. In this sense we can divide the systems in:

- **on-line**
- **semi-on-line**
- **off-line** or **postmortem**

On-line analysis is the kind of analysis that requires heavy instrumentation and a continuous, real time stream of performance data. It is rarely useful for general purpose parallel programming, mainly because of the huge amount of information coupled with limited capacity of the user to track down the events.

Semi-on-line analysis is a compromise between on-line and off-line analysis which provides a continues stream of data which, however, are not real time but have been produced in the near past of the application which is still running. This type of analysis is particularly useful for applications that may execute for a long period, days or weeks, and which we would like to analyse before the complete program is finished. For such applications, semi-on-line analysis is rather a necessity than a choice since there would be no easy way to store the tones of performance data the application might generate over a week for example.

Postmortem analysis is by far the most widely used in parallel computing not only because of its flexibility but also because it can provide useful insight with reasonable levels of instrumentation and intrusion to the analysed application. For this reason most

available tools for performance analysis and visualization of parallel systems provide mainly postmortem analysis.

Another way to classify the performance analysis and visualization tools for parallel systems is to consider the type of programming model they support and the type of parallel architecture they are assuming. Taking this approach we can distinguish among:

- **message passing**
- **data parallel**
- **general**

tools as well as

- **distributed memory**
- **shared memory**
- **general**

tools. These categories are not mutually exclusive and it is surprising to find a tool belonging to more than one. For example a tool might very well support distributed memory type of architecture which more often than not implies a message passing model and at the same time provide views and analysis for data parallel constructs which infer shared memory models. This has been very common lately, especially because of the need to provide support for HPF which although is data parallel its implementation is done on top of a message passing layer. For this reason classic message passing based tools such as Pablo are being customized to provide data parallel views [52].

1.4 Qualitative and Quantitative Characteristics

It is evident from the above discussion that there is a large variety in the performance analysis and visualizations tools. This makes their evaluation difficult since one has to come up with meaningful criteria and at the same time useful for the user of the tools.

Given the lack of standards in the area of performance analysis for parallel systems there is hardly a single denominator among the available tools. Thus, if we are to come up we a set of useful comparison characteristics it has to be relatively abstract in order to be applicable in the vast heterogeneity of the existing tools.

We have selected to address the tools we discuss in the context of the following characteristics:

- **age**: in general the age of a tool gives an indication of its current state with respect to the developments in parallel computing. By age we mean the last update of the tool. Given the rapid expansion of the parallel computing systems it is crucial for a tool to be useful to be regularly updated and revised. A tool that is continuously supported and updated will take a ***. Tools with * and ** are either older or not supported since they were produced.
- **ease of use**: generally speaking the degree of easiness that a tool possesses is quite a subjective measure. In this context we agree to give a * to a tool that has a minimum

graphical user interface. A tool that has a graphical user interface and conforms with existing standards for GUIs will get a **. A tool that has a contemporary GUI and at the same time adopts a straightforward intuitive scenario or provides intelligent support for interaction will get a ***.

- **scalability**: scalability is again a very subjective measure based on a vaguely defined concept. However, we include it because it is very important from a user point of view. A tool that is designed to cope with a fixed amount of processors or entities will get a *. A tool that is able to provide analysis on a reasonable range will get a **. A tool that can in principle scale to any number of processors or entities will get a ***. Evidently, a *** can only superficially be assigned to a tool based on its design principles and techniques employed since we can hardly test such tools for data sets coming from more than 512 entities.
- **portability**: portability is included in the list because it is very important from a user points of view since it provides a higher value to the result. If a tool can be used across several platforms the user can obtain meaningful and comparable results for the application on a range of machines. Given the distinction between the instrumentation system and the performance analysis and visualization system, we will give a * to a tool that provides portability of its visualization part. Most of the tools are built on X Windows and thus their front end can run easily on most available workstations. A tool will get a ** if the analysis model can cope with more than one parallel models. A tool will receive a *** if its instrumentation subsystem can be ported to more than one machines in a straightforward manner.
- **extensibility**: A tool that is proprietary will receive a * unless it provides for an explicit interface that allows the user to expand the analysis capabilities of the tool. A tool that provides extension capabilities which however require moderate effort will receive a **. A tool that provides for easy, straightforward extension will receive a ***.
- **software engineering level**: Since we are not able to judge the source code of proprietary tools and the methods used for their design and implementation we are going to assume that these tools have excellent software engineering levels and assign them a ***. This however need not be true. Tools that are purely implemented will receive a * and tools that are implemented on solid software engineering standards will get a **.
- **documentation**: In general a tool will get a * if it is only accompanied by a brief description and does not have any support team. A tool will get a ** if it has a set of documents including a user's guide and a reference guide and some minimum support. A tool that has a complete documentation and reasonable support will receive a ***.
- **availability**: A tool that is publicly available will get a ***. A tool that is available publicly but with strings attached will get a **. A tool that is proprietary will get a *.
- **standards**: In general a tool that has been build around existing software standards and platforms such as X11 will receive a *. A tool that is taking into account standardization efforts in the parallel development environments such as the MPI will receive a **. A tool that contributes substantially in the definition of a defacto standard or it supports such a standard will receive a ***.
- **speed/performance**: A tool that provides no special feature that would potentially make it run faster will receive a *. A tool that has provision for optimization of the

analysis will receive a **. A tool that provides for parallel execution of the analysis task will receive a *** based on the assumption that it will run faster.

- **applicability**: A tool that is able to analyse only a specific machine will receive a *. A tool that provides possibilities to analyse a whole category of applications will receive a **. A tool that provides multi-environment multi-architecture analysis capabilities will receive a ***.

2.0 Current Problems, Methods and Systems

In the last decade, along with the rapid expansion of the available parallel systems and parallel programming approaches came a large number of tools aiming in assisting the task of performance analysis and visualization. Most of these systems have been academic prototypes demonstrating ideas and methods. These systems tend to become obsolete very soon, since the groups that developed them lack the motivation and the resources to maintain and expand their original ideas. However, a number of systems have substantially influenced the field and provide the basis for future research.

2.1 Problems and Methods

Several systems base the analysis they provide on traces generated by message passing programming libraries. This kind of instrumentation is done at library level and is relatively light weight instrumentation. One of the first message passing systems to generate traces is PICL [18]. Also PARMACS [10] provides a trace facility that allows for the collection of information that can be used for analysis. The original versions of these libraries provided traces that could be viewed with ParaGraph [28]. Recently both tools have come up with SDDF [2] traces which are considered more portable and allow the use of Pablo [50]. A proprietary version of ParaGraph has also been developed that supports the SDDF format. MPI [21] which has recently emerged as the message passing standard has also provision for the generation of parallel execution profiles. However, until today it has not specified other than the interface of the profiling facilities leaving the specifics to the implementors.

The emergence of HPF [37] and HPC[4] has revived the interest for data parallel kind of analysis features. HPF (High Performance Fortran) and HPC (High Performance C) are initiatives that attempt to establish a standard for parallel programming in Fortran and C. The features they provide are mainly of data parallel programming model. One of the main features of these programming frameworks is the ability to define the distribution of various user level data structure among the processors. Evidently this distribution will largely affect the performance since it will imply specific patterns of communications. Ideally, the user would have preferred to avoid the burden of distributing the data since it implies that he or she knows how to do so. Nevertheless, the challenge exists and tools have been developed that try to cope with this. Hackstadt and Malony [22-25] attempt to build a Data Distribution Visualization (DDV) environment that will assist exactly this task. At present they support only HPF. Their main argument for introducing the DDV methodology is that the traditional displays offered by ParaGraph, Pablo or Sieve [55,56] can be effective for general parallel performance evaluation but they fail to address language specific issues. They implement a system based on the Data Explorer of IBM [35] and a software level instrumentation that provides the necessary data to do

the analysis. The data result from a semantic analysis of the code and are then transformed to a format readable by Data Explorer. The technique they present is interesting however it seems limited to the distribution of two dimensional data structures such as arrays. The problem of how to visualize and manipulate the distribution of higher dimension data structures such as multidimensional arrays, structured entities or classes is not addressed in that context. Furthermore the attractive graphical interpretations are mainly due to the abilities of Data Explorer which however is a tool that one has to buy. Consequently the approach cannot be widely used. One very interesting visual display that they introduce is the that of the Kiviat Tube, a three dimensional object where the planes depict kiviat diagrams while the third dimension accounts for the change of these diagrams over time, as shown in the picture below.

FIGURE 1. Kiviat Tube [23]

Pablo is also being modified to account for the need to support language specific analysis. Other systems have also been implemented to support language constructs [38, 40, 43, 45, 53, 54, 58]

A number of techniques has emerged that attempts to statically analyse and predict the performance of parallel applications based on a sequential source code or a parallel source code of the application. In principal it is extremely hard to statically predict the performance from an instance of the source code. The reason is that it requires substantial semantic analysis which cannot be easily generalized. It is exactly the problem that parallelizing compilers have not been able until today to address for general parallel programming. This however does not mean the techniques are useless. For example the approach of using profiles and performance estimates of sequential versions of an application to guide the parallelization process has been introduced in some systems such as the Vienna Fortran or Fortran D where interesting results are reported [3, 16].

The need to consider the performance analysis of a family of applications and the ability to use the experience available in the form of profile databases, and applications types has also been addressed as a method to perform meaningful performance analysis for application families [34, 59].

Several problems remain open and need to be addressed. One of the most challenging is the visualization and performance analysis of scalable systems that can grow into large number of processors (MPPs). A number of alternative approaches has been proposed such as adaptive graphical representation, reduction and filtering, dynamic clustering and spatial arrangement and generalized scrolling [12, 22, 48]. The problem with this methods is that although conceptually sound, it is very difficult to apply and implement.

Furthermore, the design of meaningful and useful analysis and visualization environments is still ahead. Let us not forget that the primary objective of the tools we are discussing is to help the user and not to confuse him even further. Consequently the visualizations must be intuitive and easy to interpret, otherwise they are going to be useless. ParaGraph and Pablo have gained their wide acceptance also because they built on top of sound intuitive graphic displays. The problem is that traditional displays such as animations, barcharts and diagrams do not scale easily. Perhaps the combination of such displays with filtering and clustering techniques is the way to solve the problem as we address in the second part of this report.

One of the most disturbing obstacles in the development of performance analysis and visualization tools is the lack of standards at all levels. Standards for the visual presentation, standards for trace data, standards for instrumentation. One can argue that standards are not available since their is no stable solution. However, the absence of standards makes the programming and development efforts extremely complicated and time consuming and for that slow to prototype valuable ideas. One emerging standard in the trace format such as the SDDF (which currently seems to be adopted defacto) will substantially help in this direction. Furthermore, standards such as MPI, HPF and HPC will provide even more solid infrastructure to built on.

Still, however, as D.A.Reed [52] suggests, today, despite a solid technical base, building successful, widely used performance tools remains part art, part science and part engineering.

2.2 Existing Tools

Below we present a number of tools that we consider of importance and which are widely used by the parallel computing community for performance analysis and visualization. A number of valuable surveys and collections for parallel programming tools and perfromance analysis and visualization has appeared recently [11, 19, 20, 26, 39, 41,. 44, 60].

MPP Apprentice

The MPP Apprentice [62] is available from Cray Research on Cray T3D. Using timers and counts along with compiler supported instrumentation it provides an environment for scalable performance analysis. The intrusion of the instrumentation is relatively low and thus long running applications can be analysed. The data format of the traces is proprietary. It provides a GUI, a knowledge base that support the analysis and a source code browser for the instrumentation.

TABLE 1. MPP Apprentice

Provided by	Cray Research
Main Objective	Presents performance information collected from fine-grained performance statistics with a scalable low level of intrusion
Features	Displays summary statistics, integrated with a knowledge base that suggests ways to tune the performance. Scalable measurement techniques based on summary of collected events
Interface	Graphical Interface based on X11 and Motif
Portability	Not portable since most components seem to be machine dependant

Prism

The Prism [1, 57] is available from Thinking Machines on CM-2 and CM-5. It is an integrated environment that supports debugging, data visualization and analysis. It is based on compiler supported source code profiling. The format of data is proprietary. It includes a performance adviser as well as graphical user interface. The level of intrusion is relatively low but the kind of instrumentation supported does provide for the generation of event traces.

TABLE 2. Prism

Provided by	Thinking Machines Corporation
Main Objective	Integrated environment for debugging, performance analysis and visualization
Features	Data visualization with a number of different graphical representations. Also profiling capabilities of the various system resources and intelligent performance improvement advice
Interface	Graphical Interface based on X11 and Motif
Portability	Not portable since most components seem to be machine dependant

KSR

The KSR-1 tools [9] are provided by Kendall Square Research for the KSR-1. It provides standard Unix profiling along with graphical profile viewers and supports source code instrumentation for event generation and extensive timing routines. It also has hardware support for the performance monitoring task.

TABLE 3. KSR-1 Tools

Provided by	Kendall Square Research
Main Objective	Performance monitoring of shared memory features. Performance tuning capabilities
Features	Unix like profiling tools, hardware assisted performance monitoring
Interface	Graphical Interface based on X11 and Motif
Portability	Good for Shared memory machines and thus not easily portable to DM-MIMDs

Express

The Express [17] is provided by ParaSoft and is a complete development environment available on many platforms. The part pertaining to performance analysis and visualization supports profiling, source code instrumentation and timing accompanied with visualization tools for these data. The various formats used are proprietary. Supports only its own programming model.

TABLE 4. Express

Provided by	ParaSoft
Main Objective	Provide a portable set of tools for parallel programming
Features	Profiling, source code instrumentation, timing, event tracing
Interface	Graphical Interface based on X11
Portability	Available for a wide variety of platforms

ParAid

ParAid [7] is available from Intel SSD on Paragon. It provides a complete program development environment which supports profiling and event trace generation along with instrumentation support and graphical visualization tools. The trace format used is SDDF and can be visualized using a version of ParaGraph. There is also support for system performance visualization using hardware assisted instrumentation.

TABLE 5. ParAide

Provided by	Intel SSD
Main Objective	Complete environment for application development and performance tuning
Features	Hardware supported system performance visualization, Scalable instrumentation, Use of a version of ParaGraph to Visualize traces, traces generated in Sddf
Interface	Graphical Interface based on X11 and Motif
Portability	The monitoring system seems to be system specific (Paragon), while some of the visualization tools such as ParaGraph are portable across different platforms

AIMS

AIMS [63] is provided by NASA Ames on Delta, CM-5, iPSC/860, Paragon and Convex (on top of PVM [15]). It is basically a toolkit that provides facilities to evaluate the performance and visualize the obtained execution traces. It supports graphical source code instrumentation and statistical analysis. Also has the ability to compute the intrusion level introduced by the instrumentation at a postprocessing step.

TABLE 6. AIMS

Provided by	NASA Ames
Main Objective	Toolkit that has been designed to provide for performance evaluation of parallel applications via measurement and visualization of execution traces
Features	Interactive instrumentation, Animated views like ParaGraph, statistical analysis
Interface	Graphical Interface based on X11 and Motif
Portability	Instrumentation is machine specific while the rest of the system is designed to be available on various platforms

IPS

IPS-2 [31, 32] has been developed at the University of Wisconsin and is an extensible,

TABLE 7. IPS-2

Provided by	University of Wisconsin
Main Objective	Provides an extensible interactive performance measurement tool that integrates data from a variety of sources. Supports a hierarchical approach to performance tuning
Features	Distributed computation of performance metrics, extensibility
Interface	Graphical Interface based on X11 and Athena
Portability	Portable across Unix platforms, however the programming model supported is relatively restrictive

interactive performance measurement tool available on heterogeneous networks of Unix machines. It supports hierarchical performance monitoring and parallel computation of the performance metrics. The trace data are described by a relatively flexible format, however the data collection cannot be controlled.

ParaGraph

ParaGraph [28] is perhaps the most widely used visualization tool. It analyses using a

TABLE 8. ParaGraph

Provided by	ORNL
Main Objective	Analyse performance traces based on some predefined analysis views
Features	A large number of interesting graphic displays. Relatively simple to use
Interface	Graphical Interface based on X11
Portability	The front end runs on most workstations. The monitoring traces are available to all machines that the PICL supports

number graphical views traces that have been generated by PICL. Recently it has been modified to understand SDDF too. It main power stems from the fast and intuitive displays it provides. Limitations include the lack of scalability potential and the fixed views provided which cannot be changed by the user or with minimum effort.

Pablo

The Pablo [2, 49, 50, 51, 52] system is available from the Picasso Research group at

TABLE 9. Pablo

Provided by	UIUC, Picasso Research Group
Main Objective	Provide a customizable environment consisting of a set of performance data transformation modules that can be interconnected in user-specified ways
Features	Sddf, Adaptive instrumentation, configurability
Interface	Graphical Interface based on X11 and Motif
Portability	All components of Pablo are designed to be highly portable

UIUC. Initial architecture targets include CM-2, CM-5 and Paragon (as far as the instrumentation is concerned). Pablo provides a powerful customizable environment consisting of a set of performance data transformation modules. It allows for interactive source code instrumentation and it defines the SDDF format for the trace data which has been widely accepted lately. It has data sonification abilities and provision for data immersion (virtual reality) techniques.

Upshot

The Upshot tool [29] is available by ANL and is very similar to ParaGraph. It analyses

TABLE 10. Upshot

Provided by	ANL
Main Objective	visualize execution traces generated by PICL
Features	similar to ParaGraph but each function has more generality and greater depth
Interface	Graphical Interface based on X11
Portability	on Unix workstations

traces that have been generated using PICL but in greater detail than ParaGraph.

Topsys/Patop

Topsys [5] has been developed for iPSC and Parsytec and it supports the implementation of scalable methods for performance tuning. It has features for on-line analysis.

TABLE 11. TOPSYS/PATOP

Provided by	T.Bemmerl, O.Hansen
Main Objective	Provide scalable methods for performance tuning of parallel applications on message passing systems
Features	on-line mode of operation, hierarchical approach
Interface	Graphical Interface based on X11 and Motif
Portability	iPSC, Parsytec GC

Seeplex

Seeplex [12, 13, 14, 42] has a number of interesting features for scalable visualization of execution traces. It uses PICL generated traces. It incorporates the idea of the analysis graph found in Pablo and originating in IBM's Data Explorer and Silicon Graphics AVS systems. However instead of transforming the trace data it uses a sophisticated annotation scheme that at a later stage will be exploited by the various filters to generate the various scalable views.

TABLE 12. Seeplex

Provided by	A.Couch
Main Objective	Scalable execution visualization
Features	AVS like analysis graph, scalable graphical views
Interface	Open View graphical user interface
Portability	not sufficient data to judge

PICL

PICL [18] is not a performance analysis environment. However it is one of the first

TABLE 13. PICL

Provided by	ORNL
Main Objective	Portable Instrumented Communication Library
Features	easy to use message passing interface, trace facility
Interface	library interface
Portability	available to a large number of systems

widely available message passing programming libraries to support trace generation. The traces generated by PICL can be viewed using ParaGraph.

MPI

MPI [21, 8] has recently emerged as the message passing programming standard. Its

TABLE 14. MPI

Provided by	ANL, U. of Mississippi
Main Objective	Portable Message Passing Interface
Features	Complete set of message passing functionality, Does not standardize the profiling procedures, does not specify I/O procedures
Interface	Library interface
Portability	Available to most major systems, both parallel and workstations

importance lies in the specification of the profiling interface of the library which whenever available will have substantial impact since the MPI is expected to be used widely for parallel programming.

PARMACS

PARMACS [10] is, along with PICL, among the first widely used message passing

TABLE 15. PARMACS

Provided by	Pallas GmbH
Main Objective	Portable Message Passing Programming Library for Fortran and C
Features	Relatively good performance. Not very flexible trace format
Interface	Macro calls, library functions
Portability	Available to most major systems, both parallel and workstations

interfaces (PARMACS has been used primarily with fortran) to allow for the generation

of traces. PARMACS traces can be visualized using ParaGraph. Both PARMACS and PICL are considered obsolete and their most successful features have been incorporated in MPI. Thus they are expected to have significant impact in the future.

2.3 Evaluation Characteristic tables

The two tables below rank the various characteristics presented in 1.4 for the tools presented above. Wherever we were not able to give a definite ranking we have put a n/a indication.

TABLE 16. Evaluation Characteristics (1)

Tool	Ease	Docum.	Portability	Extensibility	Scalability	Age
MPP	***	***	*	*	***	***
Prism	***	***	*	n/a	n/a	**
KSR-1	n/a	***	*	n/a	n/a	**
Express	n/a	***	**	*	n/a	***
ParAid	***	***	**	*	***	***
AIMS	n/a	n/a	***	n/a	n/a	n/a
IPS-2	**	**	n/a	n/a	n/a	*
TOPSYS	*	n/a	*	n/a	*	*
Seeplex	*	*	*	n/a	**	*
Pablo	***	***	***	***	**	***
ParaGraph	***	*	*	*	**	***
Upshot	n/a	n/a	*	n/a	**	n/a
MPI	***	***	***	***	***	***
PICL	***	**	***	*	**	**
Parmacs	*	*	***	*	*	*

TABLE 17. Evaluation Characteristics (2)

Tool	Standards	S/W level	Availability	Performance	Applicability
MPP	*	***	*	***	*
Prism	*	***	*	n/a	*
KSR-1	*	***	*	n/a	*
Express	*	***	*	n/a	**
ParAid	***	***	*	***	*
AIMS	**	n/a	n/a	n/a	*
IPS-2	*	n/a	n/a	***	**
TOPSYS	*	n/a	**	n/a	*
Seeplex	*	*	***	n/a	n/a
Pablo	***	**	**	***	***
ParaGraph	***	*	***	*	**
Upshot	**	n/a	***	n/a	n/a
MPI	***	**	***	***	***
PICL	***	*	***	*	**
Parmacs	**	*	**	*	*

Patool

In this part we discuss our experiences in designing and prototyping a performance analysis and visualization environment based on the Pablo system. First we discuss extensively the requirements that a competitive tool should address. Then we present its design and characteristics as well as the innovations introduced. Then we give a description of the interaction and the interface of the Patool which is similar to that of Pablo. A more detailed design of the Patool is to be found in the appendix of this report. There the HOOD design methodology has been used to formalize the proposed design and the necessary reengineering.

3.0 Patool Requirements Analysis

There are several issues and points that must be considered in relation to the performance analysis and visualization tool's requirements. The tool characteristics and operations are usually related to some special feature of the target system. In the general case there measures and metrics that can be displayed and visualized without necessarily knowing the specifics of their origin. Their final interpretation and understanding is left to the user of the tool. For example, general metrics such as utilization, load factor, turnaround time e.t.c. can be displayed if the data needed to calculate them are available in the trace data and the respective formulas are known and clearly defined.

On the other hand, there are specific metrics, aggregate measures, graphs and visual displays that are directly related with some special characteristic of the target system, i.e. the type of programming model, the type of architecture and so on. It is this set of data that poses specific interest for the performance analysis tool since it requires special treatment. Below we attempt to address all the issues and the points that must be considered further.

It must be noted that the main power of performance analysis tools and of the services they provide are not merely the display and calculation of naked figures but the more powerful possibilities resulting from approaches such as the ones below.

- **Application oriented analysis:** Application oriented analysis is one of the most important requirements for a performance analysis and visualization tool. This is evident, since very few of the existing systems cover this requirement, and even those which do, do so in an ad hoc, customized manner. On the other hand, the core entity of performance analysis is the application itself, since it is the object that provides the trace data and "provokes" the machine and run time environment to react and reveal their characteristics.

  At the same time, it is the most difficult to accomplish, since it is certainly dependent on a specific description or application class. Thus, the application level definition of analysis and the classification of the types of applications to consider is extremely important since it provides a methodological approach in rendering an as broad as possible application area.

- **Data transformation and analysis techniques**: It is important to note that a high degree of flexibility in the number and type of transformations implies a high degree of interaction with the user. This is evident since the user is the one that has the most complete understanding of the application under analysis.
- **Types of tool operation**: Several of the functions performed by a performance analysis and visualization tool could operate, in principal, in more than one modes. For example animation, playback or simulation could be done on-line or off-line. Of course, off-line mode of operation is the most widely acceptable since it does not require extra hardware support or permanent instrumentations and is much more flexible.
- **Data reduction and filtering**: The data reduction and filtering is one of the major problems to face. It is directly related to the scalability of the tools, and is of importance, especially when considering Massively Parallel Machines. This is true since this type of machines are expected to generate vast quantities of data and require the display of hundreds, even thousands, of objects (i.e. processors processes, tasks, e.t.c.). Furthermore, the Massively Parallel Machines are usually distributed memory machines (since shared memory machines cannot scale that much) and thus relate the performance analysis with a specific class of hardware and corresponding run time environment.

  The problem of optimal or even efficient solution to the reduction and filtering is still an open problem and thus no widely accepted solution exists. However, solutions in this direction are available and one should consider them in the context of performance analysis and visualization tools. Such solutions include:

  1. **zooming techniques** with respect to an interval appropriately defined in the performance domain of the application (i.e. time interval, process interval or memory interval)
  2. **restriction or selection** to a subset of entities or events of interest based on some predefined criterion
  3. **equivalence clustering** based on implicit relationships among the various entities (i.e. subcalculations, domain decomposition, task grouping e.t.c.).

It must be clear, however, that all these solutions require prior knowledge of the environment and the specifics of the application, since they assume a substantial intellectual interaction and consideration (i.e. they cannot be easily automated). The zooming technique is slightly more general since it only assumes some type of ordered intervals, however it is difficult to implement.

- **Cooperation with existing packages**: It is important for a performance analysis and visualization tool that is expected to be widely used to have the possibility of cooperating with other existing packages. Such packages might include other analysis tools, spreadsheets, databases or even, as described below, code profilers and parallel debuggers. This requirement could be accomplished by open design and compliance with standards, whenever available.
- **Code profiling**: Code profiling requires heavy instrumentation and produces vast amount of data. However, in specific cases where such profilers exist, the interface and cooperation of a performance analysis and visualization tool with them should be considered. Depending on the available data provided by the monitoring and instrumentation system, the performance analysis tool could provide macroscopic

(i.e aggregate) measures such as the number of procedure invocations or block executions.

- **Debugging Support**: Often performance analysis and visualization tools are used for debugging purposes, not in the source code level, but for high level errors hidden in the design of the application. Thus, it is not accurate to consider a performance analysis and visualization tool as a real debugging tool. Such tools for parallel machines are extremely machine dependent and they require excessive instrumentation. Debugging would require close knowledge of the source code of the application, and since this can only be done through the trace data, it would imply heavy instrumentation and intrusion in the observations.

  However, for specific cases, a performance analysis and visualization tool might interface to a parallel debugger.

Below we outline some basic features that ever performance analysis and visualization tool is expected to have:

- **utilization graphs** (processor count, charts (i.e. Gantt), summary profiling, concurrency profiling, utilization meters and barcharts, application specific diagrams)
- **communication graphs** (message queues, communication patterns, playback/animation, topology graphs, communication barcharts and meters, communication traffic, space-time diagrams)
- **processor/task graphs** (task topology/mapping, task profiling/information summaries)
- **application specific graphs** (application profile/summaries, allocation/mapping information, resource utilization, application specific figures (i.e. errors, convergence of solution), application specific displays (i.e. domain decompositions an data structure mapping)
- **specific analysis** (critical path analysis, behavior analysis, phase analysis)

3.1 User Requirements

It is clear that each individual user of a performance analysis and visualization tool will expect something different. This becomes evident since there are more than one tasks that one would like to perform with the support of a performance analysis and visualization tool and there more than one types of users that might use the tool. Below, we attempt to categorize these potential users and provide a scetch of their respective needs. These needs will prescribe respective tool requirements.

- **System (Architecture/Hardware) Designer**: this type of user is potentially interested in every detailed information coming out from the heart of the machine. This information will likely include registers, memory, communication network, I/O system and other crucial components of the machine. It is apparent that such information can only result from heavy hardware instrumentation and thus the needs of such users are usually covered by specialized, in-house, custom designed performance analysis and visualization tools.
- **System Software Engineer**: this type of user is probably interested in specific information coming from the run time environment. Such information might include

queue and buffer usage, allocation statistics, memory management information such as available memory, page faults and so on. Depending on the level of instrumentation, this information can be made in general available by the performance analysis tool, however, it tends to be system specific.

- **System Tool Developer**: this type of user is probably interested in data and analysis coming from the interaction of specific tools with real applications. For example communication libraries, precompilers, mappers, decomposers, static/batch schedulers and/or load balancers are all tools that interact heavily with the application programs.
- **Application Developer**: this type of user is mainly interested in application specific figures and usually requires "custom designed" information. This is true because it is not possible to have uniform information covering all the potential information for different application classes, and also since each such class has different characteristics and different data interpretation and visualization needs.
- **System Administrator/Operator**: this type of user is mainly interested in macroscopic figures of the underlying system such as aggregate utilization, failures, I/O usage, e.t.c. We are familiar with tools that offer such information for sequential systems (i.e. workstations). However, for the parallel systems is something difficult to realize.

Furthermore, it should be kept in mind that for a performance analysis and visualization tool, it is quite important to be able to cover different levels of expertise in parallel systems. Thus, a provision for different interactions must be considered. A straightforward categorization according to the level of expertise could be:

- **Novice**: a user that has no real knowledge of the parallel system. Such a user might come from a different discipline such a physics or chemistry, and merely intends to find out the behavior of the application in a parallel machine. Such a user will possibly want to perform experimentations (with respect to the application) in order to fine tune its performance (i.e. better numerical results, smaller approximation error) and thus only need an integrated environment for handling and fast interpreting the large amounts of data. Such a user will most often expect some intuitive information or visual perception of the application on the machine and might not be interested in any "confusing" information from the underling levels (i.e. hardware, run time environment).
- **Moderate**: a user that has knowledge of the parallel program (might know for example parallel programming) and who understands what is going on in the run time environment (i.e. with respect to communication, data distribution, e.t.c.). Such a user might be after improving the implementation of an application, thus looking for information from the communication subsystem, run time environment and other components.
- **Expert**: a user that has complete knowledge of the hardware and software. Such a user might be interested in every single detail that the performance data can give.

3.2 System Requirements

A performance analysis and visualization tool is a software tool that must operate in a certain environment. The actual environment of the tool poses specific requirements that need to be further analyzed.

Hardware Requirements

Most probably the hardware environment that the performance analysis and visualization tool will operate, will consist at least of a parallel machine and a front-end or host system. The front-end system that the performance analysis and visualization tool should be placed is usually a high-speed graphical workstation, with a high speed connection to the parallel machine.

It is important to realize the need of the high speed connection components since it has already been pointed out that there is a vast data transfer between the parallel machine and the front-end. At the same time the performance analysis task itself needs substantial computing power and fast graphical displays for the visualization purposes.

It should be noted that with today's graphical displays and transfer speeds it is hard to consider on-line performance analysis and visualization tools. It is very hard for the end-user to take advantage of an overwhelming high speed action taking place onto his screen. Such a system would be of interest for machine inspection in real time environments, and most certainly would require hardware instrumentation and would pose a great deal of performance perturbation and system intrusion. For this reason, "off-line" or "post-mortem" tools are considered more efficient and of more general use.

Software Requirements

The performance analysis and visualization tool will necessarily interact with a number of other software components. The run-time monitoring system and the instrumentation software are the non-trivial and important ones since the overall integration of these tools is of utmost importance for the system functionality. For this reason the interfaces among these two components must be clearly and efficiently defined in order to yield a modular tool with the desired characteristics.

The interaction between the performance analysis and visualization tool and the run-time monitoring system and instrumentation software should be through the information data and traces that they provide.

Requirements , Standardization and Characteristics of the Trace Format

One of the most crucial components and the core of the input for the performance analysis and visualization tool is the information coming from the actual execution of the application program on the parallel machine. Thus both the application and the actual traces are of importance.

Since the trace data are the interface entity, they must be described in a flexible, structured, self-describing, portable and scalable manner.

Interfaces to other Components

Interfaces to other components and external tools must also be considered. Such tools might include preprocessors, databases or other packages such as spreadsheets that need to be integrated in order to increase the functionality of the tool. For this reason, the functional dependencies with these tools must be considered and clearly defined.

3.3 General Requirements

There are several "magic" words that imply specific system characteristics within the parallel community and prevailing nomenclature. We attempt to give our interpretation of these terms in the context of a performance analysis and visualization tool, since there is no universally accepted interpretation. Below we present some of the main general characteristics that could in principle govern a system in order to make it attractive to a wide range of users.

Portability

Portability allows a familiar environment which someone already knows how to use, to be used in an otherwise alien parallel software environment.

It enables cross-system comparisons among architecture/application pairs, i.e. when analyzing and presenting equivalent application performance data on multiple, differing parallel machines, one can study the effects of system software and machine architecture in the application performance.

The portability requirement for the performance analysis and visualization tools can be met if these tools are built on top of widely available standard packages. They can take advantage of the uniformity of the "sequential" world since they are running in the front-end of a parallel machine which is in most cases a high-speed workstation.

What poses some problem in the portability of the performance analysis and visualization tool is the availability of portable performance data. This refers to data that can be used in different contexts with the same conceptual meaning and interpretation. For this reason it is important to pay special attention to the characteristics of the trace data format and manipulation capabilities. Data must be described in a self-explaining way, with a high degree of freedom following a structured approach.

Furthermore, in the case that part of the performance analysis and visualization tool functions, such as data reduction and preprocessing are done in parallel, again portability is an issue that needs further consideration.

To the best of our knowledge, no widely acceptable performance analysis and visualization tool exists that directly exploits a parallel resource for each computation needs. However, the task of performance analysis and visualization as the size of available target systems grows, it becomes itself a computationaly intensive task, which in turn might need to be done in parallel. A possible solution that might offer a degree of portability in this case, would be to built the parallel parts of the tool using an existing well

defined portability layer. It must be noted however, that this will result in less universal components of the performance analysis and visualization tool since the problems of parallel programming are inherited.

Scalability

Scalability becomes of great importance as a characteristic, mainly because of the emerging generation of Massively Parallel Processing computers. These systems deliver their services on a wide range of variable number of processing elements, starting from a few tens up to thousands. As the parallel system scales in size and performance, so must do the performance analysis and visualization environment.

Scalability of a performance analysis and visualization tool not only implies that the environment must be capable of capturing and analyzing data from a very large number of processing elements, it must also be capable of presenting those data in ways that are intuitive and instructive. Reduction and analysis of performance data from hundreds or thousands of processors is itself a computationaly intensive task and maybe some portions of the performance analysis environment might need to executed on a high performance computer. Current presentation techniques are based on presenting in a very fine grain fashion (i.e. process by process, processor by processor) the results. It is obvious that such techniques are very difficult to scale (i.e. screens are small to show thousand of processors). Moreover, they might yield quite complicated outputs which would be very difficult and time-consuming to interpret and to use. Thus, reductions and groupings might also be needed for scaling the presentation (i.e. show processors grouped under some equivalence relation, show modules of software/application which logically form a coarser part of the whole, provide aggregate and not detailed measures e.t.c.)

Extensibility

Extensibility must be carefully balanced against ease of use. If a tool's functionality is too limited, it will not be used. Conversely, if a tool is too general and does not support common cases in obvious and intuitive ways, users will abandon it in frustration. Tools should in principal be extensible in all aspects, such as:

- type of analysis (addition of new techniques)
- type of visualization (addition of new graphs and displays)
- customization (user defined events and analysis methods

Configurability

Configurability is also important since for a performance analysis and visualization tool to be challenging and attractive it must allow various, successful existing packages to communicate (i.e. common spreadsheets) and/or should be able to consider already existing software (i.e. vast collected sets of performance data).

Ease of Use

Ease of use is quite subjective requirement since it refers to each ones personal taste and sense of ease. However, a minimum set of requirements for a tool to be user-friendly and easy to use would be to offer:

- good documentation
- support
- on-line help and tutorial
- graphical interface
- color and sound
- clicking and browsing operations

4.0 Patool Design, Overview and Characteristics

4.1 An Overview of the Patool

The performance analysis and visualization tool **(Patool)** aims to assist in the understanding and utilization of *trace data* coming from a parallel machine on which an application is executing. The Patool is expected to be used for analyzing the performance based on trace data which are stored in a trace file under a predefined format. Furthermore, it is expected to present the results and data in a graphical and intuitive fashion on a graphical workstation. The tool has been designed using HOOD[1] tool and is being built on widely available packages and tools.

In general, the kind of information that is presented is based on a set of measures and metrics such as:

- a set of predefined intuitive measures that reveal information in graphical form, and which have been proved useful within the scientific community,
- a set of aggregate measures or events extracted (or compiled) from (by) the basic measures as a result of logical correlation of the available information,
- and, finally, a set of statistically defined measures.

The above list comprises a large number of information that yields rigorous insight on the machine/application behavior.

The necessity of a performance and visualization tool for parallel machines and applications has already been justified. What should be noted here is that the design and implementation of the Patool presented attempts to provide a unique tool for such a task which will be both highly competitive and easy to use. The current availability of related tools indicates a substantial gap among tools provided by the academic community and

1. Hierarchical Object Oriented Design [33]

those available on commercial machines. In the first case we observe a number of interesting features, but at the same time an inadequacy in several occasions because of their generality. In the second case, we observe high (product) quality but limited applicability, usually only on the machine they are delivered for.

Our approach attempts to fill this gap to a satisfying degree by providing a tool that, while maintaining a substantial degree of generality, it will not fail to be effective and of good quality. Special emphasis is given in the portability of the tool across several platforms so as to increase its applicability. This is accomplished to a great degree by the type of interface that has been selected with the trace data and the monitoring subsystem and the configurability of the Patool. Scalability of the Patool is seriously considered and attacked via a pragmatic approach which is based on powerful data filtering and reduction rather than on research for "fancy" graphical presentation which, until today, has proved to be disappointing. By providing the possibility of abstracting the data and selectively tracing portions of the overall computation we achieve a "functional" scalability, leaving for the unknown future the discovery of data display mechanisms that will challenge the available screen capacity limitations.

4.2 Background and Origins

Substantial work has been carried out over the years in the area and remarkable achievements prove the results of this work. We try to use as much as possible the experience gained from this work and in some cases built on top of it. Mentionable examples of previous work that we are using include the Pablo analysis system developed at UIUC [50, 51] and the ParaGraph analysis system developed at ORNL [28]. Both systems have proved to be quite popular among the scientific community and employ a number of interesting features and approaches. Both tools perform what is generally called "postmortem" analysis.

ParaGraph offers a multitude of ways that data can been presented to the end-user. These ways basically include graphical displays that present data previously gathered on a parallel machine running an application. The collection of data is based on some lightweight instrumentation hidden in the Message Passing library called PICL [18]. Overall, ParaGraph takes as input a trace file gathered from PICL and outputs a number of graphical displays of these data. The type of information stored in the trace file is statically defined and cannot change. Thus, the flexibility of data analysis is limited by the type of data collected. Moreover, ParaGraph has a limited number of processors that can handle, making it useless for analyzing data from large configurations.

Pablo gains its power from two basic characteristics. First, it gives birth to the notion of the analysis graph, in the context of performance analysis and presentation tools, giving the possibility to the user to define the type of analysis to be carried out. Second, it provides a powerful way to represent trace data, called Sddf. Sddf gives the possibility to have a dynamically changing set of information stored in the Trace file, based on some high level definition. One other advantage to be taken into account, is Pablo's powerful software construction which is completely object oriented. Still, Pablo suffers from the same problems more or less that ParaGraph does in terms of overall output. Some additional problems include the low quality, although more general, of the graphical displays, and the limited distributed capabilities that it offers.

Our design approach is based on taking from these existing solutions what seems to be interesting, and by building on top of it, provide a tool that bypasses the mentioned problems and others not mentioned explicitly here. We take advantage of a number of interesting displays and animation features offered by ParaGraph. Also we take advantage of the software architecture and the powerful features of Pablo (i.e the analysis graph and the Sddf format). Taking this approach, we our able to have a starting point which is far from zero, and which we believe is a good place to start for delivering an ambitious and challenging performance analysis tool for parallel machines.

4.3 Innovations and Extensions

In the Patool design, we have undertaken a major effort in order to provide a number of features and characteristics that will attribute to the tool the necessary advantages to compete with the existing tools To achieve this, a number of new things had to be identified and considered. We have done that not from scratch, as mentioned above, but based mainly on Pablo. In the design we follow an incremental approach by first providing a number of extensions and adjustments and then building a number of new features. We exploit the paradigmatic software structure of Pablo and use completely object oriented philosophy in the system. Also, since we adopt the Sddf format and the notion of the analysis graph as both being powerful in the context of the Patool, we are able to reuse a considerable portion of Pablo. Furthermore, we are taking advantage of several displays of ParaGraph to improve on the displaying capability of the tool. Lastly, we identify a number of drawbacks in these systems and we attempt to improve them. The points where explicit extensions and improvements have been designed include:

- The type and appearance of the displays offered by Pablo, although seemingly general, are of low quality and practical use. This is because they miss explicit correlating information with entities and events identified in the trace data and also because they are bounded to numerical values. At the same time, the ParaGraph displays are fixed statically to a maximum, something that restricts their applicability and use. We device extensions, based on Pablo displays that improve the quality and usefulness while at the same time, using the filtering and reduction mechanisms we obtain the necessary abstractions and groupings to achieve scalability.
- The fact that both tools do not allow for a multiple analysis session poses a serious obstacle in terms of productivity and operational capability. We extend the infrastructure offered by Pablo to provide for multiple analysis sessions in a natural way. This is accomplished by adding a number of features to the graphical editor and also to the management of the analysis graph. By allowing more than one analysis graphs under the same tool instance we increase the interactivity. Furthermore by providing graphic editor features that are common elsewhere (i.e. cut, paste, move) we provide for reusability of existing work and thus increase productivity.
- The fact that none of the tools provides the possibility for distributed operation in terms of processing, file sharing and session residence poses inherent difficulties and drawbacks in the overall productivity and usefulness. By extending the tool to operate on a distributed environment we provide both points to the end-user of the tool. We achieve this by reengineering the software architecture based on the power of the object oriented design and the available distributed processing capabilities in UNIX LANs.

- Finally, in all cases, the analysis applied to data is improved, both by extending the available mathematical tools but also by increasing the use of inherent properties described through the events relationship.

Although a substantial improvement over the current state of the art, the above extensions are not enough to meet our goals. We have designed a number of new developments that built on top of the extended system to combine the most useful features and to provide new ones. The extensions we designed and are being prototyped include:

- new filtering and reduction units. As mentioned earlier, they provide the means to scalable analysis via selective grouping and abstraction. The filtering unit allows for selective processing and grouping of the trace data based on event/attribute identification, i.e selection of trace data coming from a range of nodes, or selection of trace data pertaining to the exchange of information among selected participants. The reduction unit allows for the abstraction of trace data into groupings that bare some logical coherence, expected to be understood by the user, i.e reducing the events produced by a group of processes or nodes to one entity, and then analyzing that entity as a unit. This can for example allow a large machine configuration to be treated as a smaller machine by reducing the single nodes to groups of nodes treated as a single entity.
- semi-on-line analysis of data captured from a parallel application. This is extremely useful for application that run long, and most likely generate huge amounts of trace data to be manipulated effectively and timely. This improves considerably over the traditional postmortem approach and increases the usefulness of the tool for real-world applications. We achieve this by providing for overlapping of data generation/gathering and data processing. This is done by taking instances, over the time, of the trace data and process them. Although this approach does not result in real time analysis it improves substantially the "only postmortem" solution while it avoids the expensive and heterogeneous hardware requirements posed by the real time analysis.
- animation and playback units that are not found at all in Pablo and are found in a restricted version in ParaGraph. The necessity for animation and playback is apparent. Also motion features such as step and run modes, and rewind give the flexibility not available in current tools.
- new displays that are customized to specific applications and machines. This is very useful since such displays enable more efficient analysis in specific cases. For example animating communication over a hypercube architecture might prove extremely useful, since a large number of the available machines falls in this category.
- Finally, perhaps the most innovative of all the new developments is the possibility for automatic session generation. In this development, we view the use of the Patool as the means of defining the analysis tool which the user would like to have. Using the Patool, the intended result can be prototyped and experimented. When the result is satisfying we generate an independent tool that performs only the needed analysis without the burden of interacting with the overall interface over and over. In this sense, the Patool serves as a performance analysis tools generator.

4.4 Functionality

The overall functionality of the Patool has been outlined above. In this section we give a more concise description of the tool operation.

Every time the end-user uses the Patool he/she creates an analysis session. Before any session can be started, the availability of trace data and their structural description is assumed. The next step in generating an analysis session is to define the type of analysis the user wants. This is done by defining an analysis graph which represents a high-level description of the analysis procedure, including inputs, processing and intended outputs. An analysis graph can be either defined interactively or loaded from a predefined analysis graph library. At the point where an analysis graph has been defined, the user is able to select and distribute the processing of data. The analysis graph is a directed acyclic graph (DAG) and it consists of a set of units, that are represented by nodes and which perform the actual data processing, and by the arcs that denote the data routing among the processing units.

Once everything with respect to the analysis graph has been set up, the user can start the processing by binding the input points of the graph to a trace file. The User can alter the graph as many times as necessary to achieve the desired analysis. Once the shape of the analysis has been stabilized the user is able to generate a stand-alone version of the Patool, i.e a new tool, that will routinely perform the stabilized analysis in the feature. Viewing it this way, the user can generate as many "ParaGraphs" as necessary to get the job done.

The User can define whether the graph is supposed to work on a static fashion (postmortem) or in a windowing fashion (semi-on-line).

The definition of the analysis graph consists in selecting the units that are going to do the processing, interconnecting them in a logical sequence and configuring their operation. The ability to graphically manipulate the analysis graph allows to reuse previously generated analysis graphs as part of the definition of a new one.

The overall interaction with the tool can be done in a distributed environment. The User can request the distribution of the processing of the graph in a subtree level.

4.5 Components

A general observation of the implementation structuring of the Patool is that it is completely consisted of objects. Each module usually is the home of more than one object, providing a set of operations. The purpose of each object in most cases becomes apparent from its context. Whenever this does not hold, we make an explicit comment. In general, all objects provide operations for creation and deletion. These operations are designed in such order, that when an object is generated, a data structure along with the operations that manage it comes to birth. When it is deleted, it simply seize to provide any operation. Notice, that we can have more than one objects of the same type, thus we are able to duplicate or distribute their functionality.

The overall design of the system can be regarded as a client-server arrangement. The Manager module is the server of the system, the Toolkit and FdSuit contain clients of this server, and the Access and User Interface contain the methods with which the clients interact with the server.

User Interface

The User Interface module contains a number of operations that provide the possibility to interact with the system. It is comprised of four modules, the Control, the Display, the Tools and the Help.

1. **Control**

   The Control module contains operations that are used to create the various menus, manage the Patool resources (pertaining to the look and feel of the tool) and to start up the Patool.

2. **Display**

   The Display module contains operations to create and manage all the necessary dialogs and interaction points with the user. It provides for three different types of dialogs.

3. **Tools**

   The Tools module provides operations for interacting with other tools via the Patool. At this point its operations are not completely determined since there has been no decision on which tools to integrate. This is the interface point of the Patool that needs to be exploited at a later phase. Currently, only a number of foreseen operations has been prototyped.

4. **Help**

   The Help module contains all the operations necessary to generate and interact with Patool help system.

Manager

The Manager module contains the operations that built up the actual kernel of the Patool. In this sense, this module can be considered as the server and the functional units that are described later as the clients of this server. It is comprised by three different modules, the Dispatch, the Run Time and the Data.

1. **Dispatch**

   The Dispatch module, contains the operations that are used to request general service from the run time system.

2. **Run Time**

   The Run Time module contains the start up operations as well as the main service process that forms the heart of the Patool.

3. **Data**

   The Data module contains the operations that are used to service the general requests to the various processing units and to the environment of the system.

FdSuit

The FdSuit module provides two processing units and a set of operations to manipulate graphically the analysis graph. The modules are the Filters, the Reducers and the Graphical Editor.

1. **Filters**

   The Filters module provides the filtering processing unit which allows the selection of events and data ranges from trace data. As mentioned earlier, this unit is the basic vehicle for obtaining scalability in the analysis. The filtering is done on those data that contain types of information that can be fully ordered. It provides also for the definition of predicates as a means to filter the data.

2. **Reducers**

   The Reducers module provides the reducing processing unit which allows to abstract over the trace data. This is accomplished either by range selection or by predicate definition and the result is a new entity that groups the information of the selected entities. For example, when we want to treat a range of processors as a single processor, we specify the range of these processors to map to a new processor which now becomes the entity of reference. This technique is powerful in abstracting large configurations and complex applications in a way that we can tackle the analysis of their behavior more efficiently.

3. **Graphical Editor**

   The Graphical Editor module, provides a number of operations to manage the analysis graph. It provides operations for managing nodes, edges and the display and manipulation of these two objects in conjunction with the analysis graph.

Toolkit

The Toolkit module is the home of the majority of processing units that are used in the analysis graph. These units are structured in three different modules according to their intended functionality. A fourth module is included for extensibility purposes and currently provides no processing unit. Additional processing units extending the functionality of the system are to be added there. Each processing unit is implemented as an object that provides a set of operations. The types of operations provided by the processing units are, as expected, quite similar. All of them have creation and deletion operations. All of them have run operations which implement their run time behavior. Most of them provide for storing and loading their configuration parameters from files, thus allowing the creation of analysis graph libraries which have readily configured modules. Also most of the units have a number of secondary operations such as *init*, *configure* and *ready* which assist in their run time management. A large number of units has been directly adopted from the functional units list found in Pablo with minor modifications and a number of units has been created from scratch.

1. **Transform**

   The Transform module contains two main modules. The Mathematical and the Synthesize module.

   The Mathematical module contains a number of processing units that perform mathematical transformations on the data.

The Synthesize module contains three different modules which provide operations for restructuring the available data. The synthesize array provides operations that allow the creation of multidimensional entities from a collection of scalars. The synthesize vector provides a number of operations for creating vectors from a collection of scalars and the synthesize coordinates provides a set of operations for creating sequences of ordered pairs (i.e. cartesian product) from a collection of scalars. All these aggregate data structures are of use in the various presentation and display processing units.

2. **Present**

   The Present module contains a large number of processing units that are used to display the data. All these processing units take as input either immediate data from the trace file, or data that are the intermediate result of preceding computations.

3. **Display**

   The Display module contains three basic modules, the Playback, the Animation and the Zooming. Each one of them provides a processing unit that generates the respective output. The animation and playback differ in that the playback is done on a general diagram of a hypothetical machine, while the animation simulates a predefined architecture. Currently the only two foreseen are hypercubes and arrays, while the playback is done in a ring like (fully connected) machine diagram. The Zooming can be applied as a preprocessing step to animation or playback in order to provide groupings of processor that will appear. This allows to focus on a particular portion of the application or of the machine.

4. **Extent**

   Currently it provides no operation. It only provides the templates and methods that will make easy the extension of the unit library.

Access

The Access module contains the implementation of all the necessary operations to manipulate the trace data. The capabilities offered by these operations are those that give birth to the power and flexibility in moving data around, from the trace files to the analysis graph processing units, among the processing units, and from the processing units to the output units. These operations are implemented within the Query Support module.

Also operations that read the interpretation procedures encoded in the Trace Data module, allow for the conversion among different trace data formats. These operations are implemented in the Conversion module.

The Database module currently is empty. Its purpose is to provide an Interface to a future extension in the Patool that will allow for reading Trace Data formats that are stored in Databases (i.e. SYBASE, ORACLE or INGRES) and not in plain files.

1. **Query Support**

   The Query Support module provides operations for four different kinds of entities. Dictionaries, Descriptors, Iterators and Pipes. Each entity has a different purpose, but all of them pertain to the manipulation of the trace data. They are named after the structures suggested by the Sddf format. Dictionaries provide operations for manag-

ing sets of structures, records and tags. Structures contain records, and records are identified in terms of tags. Each individual structure is assigned a descriptor. Each individual record consists of fields. Each field defines some attributes. Thus the set of operations provided by the three modules, Dictionaries, Descriptors and Iterators is combined to manipulate the trace data in a hierarchical fashion.

The Pipe operations assist in moving data among the various input, processing and output units of the analysis graph.

2. **Convertion**

   The Convertion module provides operations for converting among different trace file formats and storage versions. There are operations for converting between PICL and Sddf and also versions for converting among Ascii and Binary encodings of Sddf formats. The third type of operation, between Sddf and Other format is there for extension purposes and currently provides no implemented operation.

3. **Database**

   Currently, the Database module provides no operations. It contains only templates and methods that model the integration with a DBMS system that would support the organization and handling of trace data.

Trace Data

The Trace Data module incorporates the definition of the trace data structures imported in the system. The information in this module is crucial in the sense that it is the only place where the structure of the trace file is recorded along with the detailed description of the individual information to be found. The module is divided in two submodules. The first provides the definitions of trace data in Sddf format. The second defines the interpretation structure to be used to import in the Patool trace data that have been generated by some other tool.

1. **Sddf Specifications**

   This module contains the Sddf trace data source descriptions. The operations provided by the Access module, allow for reading and deciphering the information in these data structures.

2. **Alien Specifications**

   This module contains the interpretation procedures that must be followed when reading trace data that are provided by other tools in some alien format. For each alien format supported, an Sddf description of the interpretation procedure is included.

5.0 Patool Use, Interaction and Interface

In this chapter we attempt to give a practical description of the Patool features and characteristics.

Repeating the general idea, the Patool is aimed to be a software tool that is used for performance analysis and visualization of parallel application/architecture pairs. Thus, it is

natural to assume that the user of the Patool is interested in analyzing a given application that has been running in a parallel machine.

5.1 Use of Patool

The first and most important piece of information assumed by the Patool is the availability of trace data that contain information about the application that runs on the machine and possibly information regarding the machine status while running the application. The Patool accepts trace data in Sddf format.

The second piece of information assumed by the Patool is the availability of an analysis scenario. That is to say, the knowledge of what kind of information needs to be analyzed, in what sense, and how it should be presented. This information is provided in the form of a configuration, which is nothing else but an analysis graph with source the trace data and sinks the various desired outputs of the analysis. A configuration may be either available to the user (i.e. it has been previously specified by others and is available in the form of a library) or it can be interactively specified using the Patool. Normally, a routine analysis configuration that is done over and over will be defined only once and used with different sets of data. On the other hand, an analysis configuration that is aiming in some customary data will be defined interactively and maintained accordingly.

As mentioned above an analysis configuration is nothing else but a directed acyclic graph with a set of source nodes and a set of sink nodes (i.e. a network). The source nodes represent the raw input to the analysis and the sink nodes represent the output of the analysis. The internal nodes of an analysis configuration represent intermediate "computing" steps, that are necessary either to transform the raw data in a convenient data structure or to compute some new data as a function of the raw data.

Having in mind the above and assuming that the user needs to define (i.e. there is no available configuration for his/her problem) a configuration we can summarize the steps that need to be taken as follows:

- Decide on what the raw data are, i.e. choose the trace data file(s) to be used. The trace data might be available in a postmortem fashion or in an semi-on-line fashion.
- Decide on what the analysis output should be, i.e. a bargraph depicting message traffic, an average of the message size among the messages exchanged between a set of nodes, a piechart denoting the percentage of messages originating from each of a set of different nodes, e.t.c.
- Identify the intermediate steps that are necessary to transform (or process) the raw data in order to have the necessary output,
- Specify, through interaction with the Patool, an analysis configuration that matches the above decisions.

The definition of the analysis configuration that matches the kind of analysis the User wants to do can make use of a large number of functional units (i.e. input, output, transformation, computing units) available within the Patool. The definition of the graph is done easily, in a graphical manner, by creating nodes and linking them together in a fashion that is a mapping of the processing the user has already identified.

The above interaction is the core interaction that a user has with the Patool. Once an analysis configuration has been created (or loaded if using an already available configuration) then the graph is run.

Running the analysis configuration effectively means start feeding raw data in the source set of graph nodes and receiving output information on the output nodes. The multitude of the output information is meant to provide different possibilities that will reflect the needs of the application that is being analyzed.

A complete list of the available functional units can been found in the Patool design document. Here we only mention briefly some classes of them:

- File input units: used only as source nodes to denote a trace file (raw data) input point to the analysis configuration,
- File output units: used only as sink nodes to denote a "transformed" data output. Reasons for using such a unit might be for example the need for "preprocessing" of the trace data and a generation of an intermediate data object in Sddf format (e.g. the initial data but filtered to include only a range of processors and their respective information),
- Structural transformers: units that given a sequence of raw data generate multidimensional structures of the data which can be later used more conveniently,
- Graphic units: used to output the collected information in a graphical way, i.e. in the form of a bargraph or a piechart, but also used for simulation of a sequence of events, such as for example the animation unit.

5.2 Interaction with the Patool

As already mentioned, the interaction with the Patool is necessary at various stages of an analysis session. Reasons for interacting with the tool might be:

- housekeeping of files and saved configurations,
- analysis configuration definition, loading and updating,
- data format conversion (i.e. from Sddf to PICL),
- analysis session (i.e. running an analysis on a on a specific instance of data).

The above reasons constitute the majority of interaction probes with the system. In the previous chapter we saw an extensive description of an interaction needed to accomplish the analysis definition. Here we will attempt to correlate the abstract actions with specific, in the context of Patool, actions.

The Trace file

As already mentioned, the most crucial piece of information needed to initiate an analysis session is to have available some data to analyze. These data, referred to as trace data are either available as a file somewhere in the file system (postmortem analysis), or are to be provided via an on-line connection with the monitoring subsystem (semi-on-line analysis).

The trace file information appears in two ways within the Patool. One is when the trace file is to be converted (if the user selects so) and the other when the file is to be used for processing. In all other cases, its existence is assumed but there is no explicit reference to it.

In the first case, where the user wants to convert the trace data file from one format to another, he/she interacts with the general menu. By selecting (Tools)(Converters...)(Xxxx)[1] the user is provided with a file selection dialog in order to define the source and the destination. When both file names have been specified, the conversion takes place.

In the second case, where the trace data file must be used for actual processing, the interaction takes place during the analysis graph definition. When a source in the analysis graph is defined (via the module selection dialog), it must be configured with information that either specifies a static trace file, or an on-line connection analysis data feeding.

The above two cases are the only two when the actual trace data file must be referenced by the user. However, there is a large number of tasks that involve the trace data file which are done invisibly from the user and are performed automatically by the system. Let us briefly mention some of them:

- information decoding,
- data interpretation,
- data dictionary loading and update,
- data generation (in the case we are dealing with an output processing unit).

The trace data themselves can come either in descriptive (ascii) form or in compact (binary) form. The user can change the representation using the (Tools)(Converters...)(Xxxx) selection. Data within the Patool system are handled in compact form, thus, when the initial file provided is in ascii, it is automatically (dynamically) converted into binary and used, for reasons of efficiency.

The Definition of the Analysis

As already mentioned, one of the most important tasks the user must carry out with the help of the Patool system before doing any actual analysis, is the definition of an analysis scenario. In the context of the Patool this is done by defining an analysis (configuration) graph. This graph is going to be used for the actual analysis and processing of the trace data.

The analysis configuration may have already been provided in the form of a configuration or a layout, in which case the user need only worry of filling in the details. The available configurations and layouts may have been produced by other users and are to reused, or may have been developed by any other person to facilitate the analysis pro-

1. The notation indicates the selections that need to be made in the pulldown menus of the Patool. Whenever a name consists of x's it denotes that any selection at that level applies.

cess. In the case of an existing configuration or layout the user needs only to specify his/her selection by selecting (File)(Load Configuration...) or (File)(Load Layout...). The difference between a configuration and a layout is that the first comes with most of the information regarding the units already set up, while the second comes only with the design (blueprint) of the analysis graph, leaving the configuration of the information for the user that is going to use it. The layout can be viewed as a sort of a template for an actual configuration.

More interesting is the case where the user must actually specify the analysis graph and configure it on his/her own. In this case, the first step to take is to come with an outline of the intended analysis (i.e. a blueprint of the graph). Knowing the available processing units, the kind of information available in the trace data, and the kind of analysis to be performed he/she must decide the outputs expected from the analysis and the way these outputs are to be produced from the available input. For example, it might be the case that a simple barchart depicting the size of the maximum message sent from a given node, against all nodes, is desired. Assuming that the trace data contain information about the size of message and the origin of the message, then a graph to perform the above task would consist of a number of processing units, which have to be decided by the user. For example, an input unit will be necessary to obtain the data from the file (or the on-line connection), a transformation unit could possibly be used to transform the information in vector form and a bargraph unit to display the information. Of course, this the simplest among all possible tasks; in the more interesting cases a larger number of intermediate steps would have to be defined, sometimes resulting in quite complicated analysis graph. This is why the user should be prepared with an analysis scenario at hand, in order to facilitate the actual analysis graph definition.

A possible sequence of steps for the definition of the analysis could be:

- obtain the trace data and study the information they provide,
- define the intended output of the analysis, i.e. a metric to be computed, a graph to be produced,
- come up with a strategy to obtain the outputs given the inputs using the processing (functional) units provided by the Patool.

The Analysis Graph

The definition of the analysis graph is the core task to be carried out using the Patool system. Once an analysis graph has been defined, we have an instance of performance analysis and visualization, that produces a set of outputs, when provided with the assumed inputs.

Let us see closely what the actual interaction with the Patool involves in order to define an analysis graph. Having decided on the analysis scenario, the user must interact with the tool in order to realize the conception he/she has come up with, and do the analysis. The steps, in sequence are the following:

- (if this action is taken, then we skip all other actions and go directly to configuration) Load a configuration or a layout by selecting (File)(Load xxx...)

- Start selecting the modules that are going to participate in the analysis graph. This is done by selecting (Module)(Add...) from the main menu. Upon selecting this option a dialog comes up with all the available choices. The user must specify which module (functional unit) to add.
- Having selected all the modules to participate in the analysis, the user must connect them in a way that will indicate the sequence (path) which the data are going to follow. This is done by selecting (Module)(Connect...) and specifying the two modules to be connected. When the connections among the modules have been defined, the graph will be topologically sorted in order to device a sequence of evaluation (processing) of the data. Some subtle points include for example, that the input modules cannot have incoming connections; if something like that is attempted a warning is issued and the action is discarded. The same holds for output modules; they can not have outgoing connections, since that would be senseless, and thus such an action is signaled by a warning and is discarded. Also self-referencing connections are not allowed since they are meaningless in this context, and thus they are signaled by a warning and are discarded.
- Having connected the various processing modules, the next step is to configure the graph, which means to specify a number of information that will customize the various units for task at hand. There are three different configuration modes:
  1. configuration of the whole graph (by selecting (Configure)(Graph...)),
  2. configuration of a single functional unit (by selecting (Configure)(Module...)),
  3. and configuration only of the module parameters of a module (by selecting (Configure) (Module Parameters...)).
- (In case the first selection was taken) configure only the graph by selecting (Configure)(Graph...). This will automatically ask only for the missing pieces of information to complete the configuration.

As described above, the task of defining the analysis graph is easy and straightforward, given that the user has some analysis scenario in mind. In case some mistake is done, the user can delete the "error" module by selecting (Module)(Delete...) and specifying the module to be deleted. In the current version of the prototype, this is the only "graph editing" the user is allowed. In the subsequent releases, the editing capability will be substantially increase by the graph editor, which will provide a full set of graphic editing functionality.

Highlights of Analysis Components

As we mentioned above, one of the tasks during the definition of the analysis graph is to select (and configure) the various participating functional units. Each time the user selects a functional unit, he/she is provided with an instance of the corresponding class of functional units, and a schematic with the unit's name is drawn in the area of the analysis graph. Note that the name can be customized by the user or can be left with the default value provided by the system.

A file-input functional unit is used in all the configurations since it is the hook to the trace data to be processed. In this sense, such a unit is a prerequisite. A number of other units can be used to built up the configuration of the analysis graph.

One very important functional unit is the filter-and-reduct functional unit. This unit is used to perform "select" and "project" type of operations in the input data, so as to reduce the size, the dimension, the variation or any other characteristic. The filter-and-reduct unit applies a logical operation (selected among a set of predefined logical operations) between its "key" and its "value" input. The function per se is primitive. However, by cascading a number of filter-and-reduct units we can produce any complicated filtering function on the stream of data. Using the set of predefined logical operations we can compute any logical function, and thus any selection or projection operation. The notion of the key and the value is abstract, and the user can feed in these inputs any relevant (comparable) field of data. Moreover, the value field can be interactively defined to contain any constant value that the user desires.

Let us provide some examples of the possible use of the filter-and-reduct unit in order to manage the size of the trace data information.

Example 1: Assume that we want to analyze the communication patterns and characteristics of a given application and among other things we are interested to know the number of "large" and "small" messages exchanged. Assume that we separate between a small and a large message by comparing its size to a threshold value *e*. We can design the above task to use two filter-and-reduct units along with two counter units attached to each other. In both we are going to connect the file input unit. In its one we are going to define as value the threshold value e and as key the message size field of the trace data record. However in the first we are going to ask (i.e configure the filter-and-reduct unit) for the key value to be smaller than the value while in the second we are going to ask for the key value to be larger than *e*.

Example 2: Assume that we want to minimize the traffic in some specific portion of the analysis graph (a subtree of the graph) in order to speed up the analysis process. By studying our data, we have decided that only those data records for which the "time" field is older than *t* should be considered. We can do that, by adding a filter-and-reduct unit as a preamble to the root of the subtree of interest, identifying the key value as that of the time field of the record, and putting in the value the value *t* of the threshold time.

Example 3: Assume that we want to minimize the number of individual processing elements (nodes) of the parallel machine we are about to consider, by grouping the processors in groups of *l/group*. Thus we are going to study the events of only *#nodes/l* different processor sets. We can do this by defining an array of filter-and-reduct records, cascaded in couples to check the upper and lower range. We are going to need for this purpose *l* pairs of filter-and-reduct units, as a preamble to our actual analysis graph. Each pair is going to provide stream of data concerning a specific group of processors.

Example 4: Assume that we want to create a new stream of data, that is going to be a subset of the initial stream coming from the file, but with some fields extracted, and the rest of the fields containing values only within a given range. We can do so by attaching a string of cascaded filter-and-reduct units to our file input unit, and configuring them according to our intentions.

Another set of very important units, are all those units that provide information about the analysis in a visual manner. Such units include bargraphs, piecharts, leds and others,

and also animation and playback units. Each of these units can be configured in order to be customized for the task at hand.

A functional unit among those with visual effects with a predefined task, is that of the animation. An animation functional unit, provides the means to visualize the sequence of a set of events, by keeping the sequence with which the events occurred. The animation unit, "animates" the actual machine via a schematic, which is a drawing containing a set of nodes, identified by their number. Every time an event is seen that correlates a source node with a target node, for example a message sent event from node *b* to node *c*, a line is drawn in the schematic between node *b* and node *c*. This pattern continues until all events of interest have been animated, or until the user decides to stop the animation.

As already mentioned, any number of output units can coexist within the same analysis graph, as long as they are connected to some stream of data, and of course as long as the physical capabilities of the screen are not obscured.

Running an Analysis Session

Once everything has been set up properly the user can run the analysis session. To do so he/she must select the (Run)(Xxxx...) main menu option. The options available for running the analysis session are:

- (Run Panel...) which produces a dialog allowing for the step execution of the graph,
- (Run Graph Once...) which runs each functional unit in the graph only once, i.e. only one record of data,
- (Run...) which runs continuously the analysis graph,
- (Stop...) which suspends the graph execution,
- (Speedometer...) which provides a scale to control the running speed of the graph.

The graph can be run over and over again; each time the user wants to rerun the graph, before selecting some run option he/she must select (File)(Restart Execution...).

The actual algorithm for running the analysis graph is as follows. The graph, as mentioned earlier, has been topologically sorted after it was configured. This order of functional units is the one used by the system to process step by step the information. A complete cycle of running the graph is done when all the modules have been run at least once. Then a new cycle starts by obtaining the next packet of trace data for processing. The notion of a complete cycle through the graph is reflected in the run options available, especially those providing for stepwise execution.

As the graph is being run the various outputs are updated; actually only those with some visual effect can be observed, since they are the ones that have separate windows. An output that its task is to write some transformed version of the input data in an output file cannot be observed.

5.3 Interface of Patool

It has already been mentioned that the interface of the Patool is a window-based interface. All actions can be taken by using the pointer device, limiting the keyboard use only for literal or name definition.

The whole GUI interface of the Patool has followed the Motif style guide as much as possible, and thus the window configuration and most of the selections are no surprise.

In this section we are going to describe in detail the Patool interface and the options available to the user.

We can divide the interface of the Patool in to the following parts:

- main menu and pull down menus,
- analysis graph definition,
- and component configuration and set up.

Below, we describe each one separately in terms of functionality and use.

Menus

Let us take a close look in the menus of the Patool. The menus consist of the main menu of the system and the various menus that come up as dialogs, having either the form of fill-in text, or exclusive selection or simply selection.

1. The Main Menu

   File: The file menu option contains all those operations that have to do with file operations as well as some management operations:

- Delete Graph: delete whatever analysis graph appears in the working window.
- Load Layout: load a layout of an analysis graph that is stored somewhere in the file system.
- Load Configuration: load a configuration of an analysis graph that is stored somewhere in the file system.
- Save Layout: save the layout of an analysis graph that appears in the working window.
- Save Configuration: save the configuration of an analysis graph that appears in the working window.
- Restart Execution: reset the input data unit and prepare in order to run the analysis session that appears in the working window from the beginning. Applies only to postmortem analysis.
- Quit: quit the Patool.

  Run: The run menu contains the various options for executing an analysis session.

- Run Panel: brings up the Run Dialog that allows the user to execute the analysis session on a unit by unit basis.
- Run Graph Once: run the analysis graph only for one cycle.

- Run: run the analysis graph continuously.
- Speedometer: provide a scale for the user to able to control the speed of the graph execution.
- Stop: freeze the execution of the analysis graph.

Configure: The configure menu contains the various options for configuring the graph and the modules.

- Graph: configure, iteratively, the whole analysis graph. This involves configuring each unit of the graph until all of them have been configured.
- Module: configure a single module, instead of the whole analysis graph.
- Module Parameters: configure only the customization parameters of the module of the analysis graph.
- Defaults: configure general settings of the Patool.

Module: The module menu contains the various options for creating, deleting and connecting units in the analysis graph.

- Add: add a processing unit in the analysis graph that appears in the working window.
- Connect: connect a processing unit to another processing unit. The connection is directed, that is to say, the unit specified first is the source and the unit specified second is the sink.
- Delete: delete a unit from the analysis graph in the working window, given that the output of the unit is not used as input for some other unit. If this is not true a warning is generated and the request is discarded.

Tools: The tools menu contains the various options for the ad hoc tools added to the Patool.

- Ascii2Binary: convert from ascii to binary Sddf.
- Binary2Ascii: convert from binary to ascii Sddf.
- Sddf2Picl: convert from Sddf to PICL.
- Picl2Sddf: convert from PICL to Sddf.

2. Graphic Menu Bar

The graphic menu bar has been designed for speedy access to frequently used functions of the Patool main menu. This bar resides horizontally below the information labels and above the analysis graph window. It hosts a number of graphic buttons that are activated upon selection. The function that is provided by each button can be induced by its respective iconic. The buttons are divided into four groups (by horizontal spacing), grouping different functionality.

The first three graphic buttons (from the left) provide operations related to a single analysis module. These operations are add, delete and configure.

The next three graphic buttons provide operations related to the analysis graph as a whole. The operations provided are delete, configure and load.

The next three graphic buttons provide operations that are related to the execution of the graph. The operations are run and stop. Also there is a button for the quit operation that when selected quits the Patool.

The last (rightmost) button is provisionally available for "toggling" the position of the graphic menu bar from horizontal to vertical and vice versa.

A small number of additional graphic buttons is to be added for further extending the speedy access to frequently used operations. Possible candidates are the restart and connect operations.

3. Module Selection Dialog

   The module dialog appears when the user selects to add a new unit to the analysis graph on the working window. The dialog provides a set of lists of available units, among which, the user is supposed to select the one to be added. The units are divided in conceptual categories for easier interface. The user can either select to keep the automatically generated name for the instance to be created, or to provide his/her own.

4. Run Dialog

   As already mentioned, the run dialog appears when the user selects the Run Panel option. This dialog provides a number of options that govern the way the graph is going to be executed.

5. Defaults Dialog

   The defaults dialog appears when the user selects the configure defaults option form the main menu. This dialog gives the opportunity to the user to define a number of characteristics that affect the operation of the Patool.

6. Record Binding Dialog

   The Record binding dialog appears during the configuration process, either of the graph or of an individual module (assuming the inputs of the latter have been specified). It essentially helps the user to define the way in which the various fields of the input are going to be used to feed the inputs of the unit and the outputs, if necessary, of the unit.

7. Input Binding Dialog

   The input binding dialog comes up during the configuration phase, and provides the opportunity to the user to define the sources from which a given unit is going to assume its input.

8. Output Binding Dialog

   The output binding dialog appears during the configuration for those units that are used in internal nodes, such as the transforming units, and provides the opportunity to the user for defining the way the fields of resulting data records are going to be compiled from the results of the unit.

9. Individual Units' Configurations Dialogs

   A large number of dialogs exists for the user to configure (customize) the various processing units that can be used in the analysis graph. The simplest dialog of all is the of the file-input unit which asks for the name of the trace file to use for data importing. Dialogs for other units include selection among a set of predefined options, as is the case for the filter-and-reduct unit. Others, such as the one for a bar-graph unit, require the user to provide information on what to appear on the axis, the range of possible values to treat and so on.

5.4 List of Analysis Units

Below there is a list of analysis units that can be used in the definition of the analysis graph.

- **FileInput**: provide the interface needed to denote a source of input for the analysis. This unit is, under normal circumstances, bounded to a trace file. The file must be in Sddf format.

  FileOutput: provide the interface to write an output file of data in Sddf format, as a result of the analysis. The structure of the data to be written is completely customizable, and can, virtually, be anything; from simple statistics on the trace data to transformed versions of the trace data.

- **SynthesizeArray**: process the input data and using the information construct aggregate data structures (2 dimensions) that collect the data (or some part of it). The resulting data structure is an array. The accumulation and storage is done automatically.

  SynthesizeArrayElement: similar with the above, but collects the data only for one element of the aggregate data structure.

  SynthesizeVector: similar with the above, but the resulting aggregate data structure is a 1-dimensional array (i.e. vector). The structure formulation is done automatically.

  SynthesizeVectorElement: again synthesize an aggregate data structure in the form of vector but for one element of the structure only.

- **FilterAndReduct**: process the incoming data according to a predefined logical predicate and filter out those that fail to satisfy the predicate.

  FilterRecords: process the incoming data types and separate them by discarding or advancing a specified subset of them.

- **Animation**: process the incoming data and depict them, animating a predefined relation (i.e. send message, receive message node pairs) on schematic that "resembles" the actual machine. It provides two schematics, one ring and one 2-dimensional array. The configuration process allows to select among these two options for the schematic.

  BarGraph: process the incoming data and display them in a bar kind of representation. The prespecified parameter(s) is used to index the bargraph, and the incoming values are used to draw the bar.

  Bubble: accepts as input a 1 or 2 dimensional array with values, and represents these values with colored cycles whose diameters vary with the magnitude of the value, on an appropriate dimension grid.

  Chart: accepts as input scalar values and draws them with respect to time on a continues line.

  Contour: accepts 2 dimensional data and draws them on a 2 dimensional grid, with values represented as contours and their magnitude used to identify the shape of the contour. The contours are usually convex.

  Dial: it takes as input a scalar and represents its value by positioning a radius on a cycle. Starting at zero in 12 o'clock position the magnitude is represented by the clockwise offset, in degrees, of the drawn radius.

Kiviat: accepts a vector of values and draws them on a kiviat diagram. The kiviat diagram is similar to a barchart wrapped around a cycle. For example, the cycle is divided into sectors, as many as the number of dimensions of the space in which the vector belongs. The magnitude of a value is drawn from the cycle center to the circumference.

Led: represents its input value as stacked layers of different color. Each layer covers a range of magnitude. The actual magnitude of the provided value divided by this range gives the number of layers used.

Piechart: accepts a vector of values and represents each value as a piece of a pie (i.e. a cyclical sector on a cycle). The length of the sector varies with the magnitude of the value.

Scater3D: accepts as input an array of triples (representing 3D coordinates and draws the values in a 3 dimensional cube. The location varies from the origin of the cube along with the magnitude of the values.

- **BinaryMath**: accepts two operands (of the same dimension with maximum dimension 2) and performs a selection of basic binary mathematical operations to provide its output, which is the same dimension with its inputs. For those cases that the operands have different dimensions (i.e. scalar-vector multiplication) the result dimension is the higher among the input dimensions.

  Logarithm: accept a scalar as input and provide as output its logarithm (with base 2, 10 or *e*).

  Power: raise a predefined base value to the value of the input and provide the result as output.

  ReductionMath: accept a vector or an array of values and perform a selection of mathematical operations to reduce the input to a single value which is provided as output. For example, this way we can compute a summation or a product or any other norm of the structure.

  UnaryMath: accept as input a scalar, vector or array and perform a selection of unary mathematical operations to produce the output which is going to be of the same dimension as the input.

5.5 Snapshots of Patool

FIGURE 2. Patool Interface

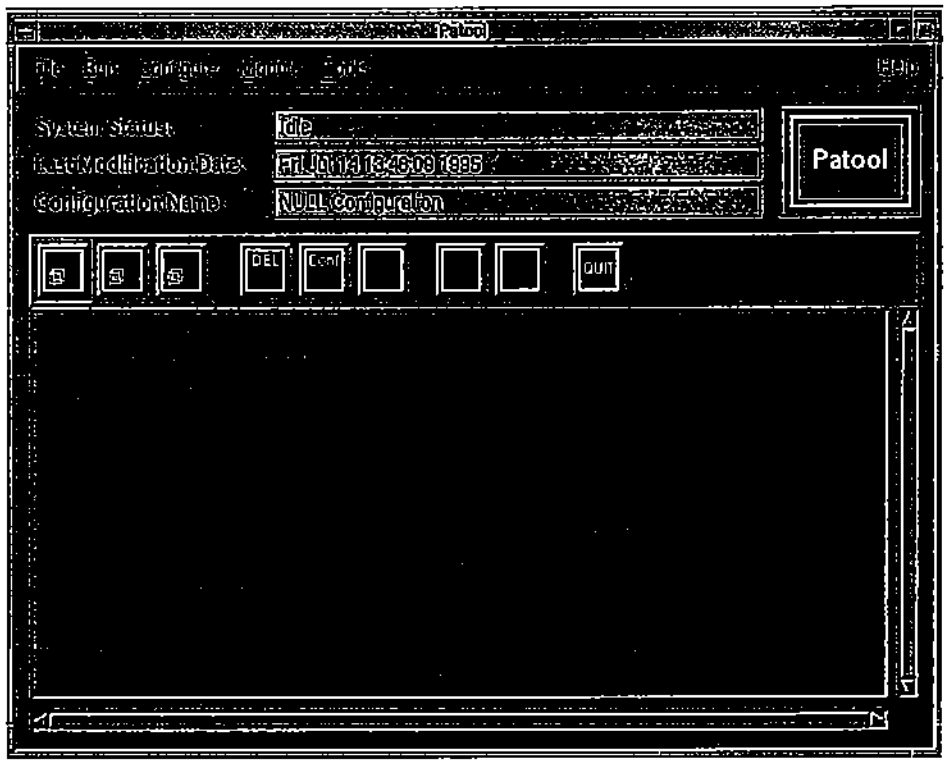


FIGURE 3. The Delay (speedometer) dialog

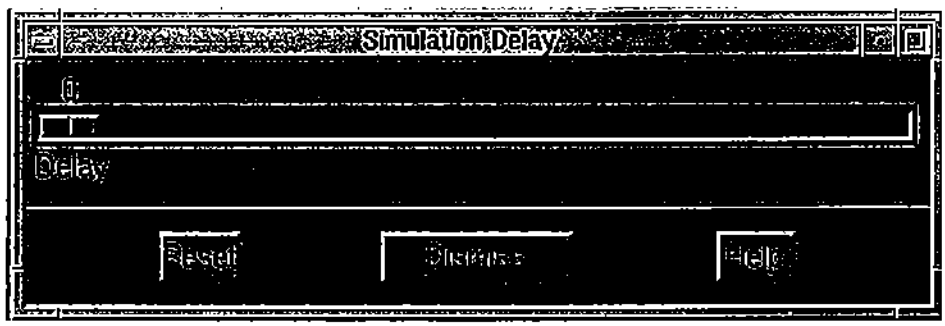


FIGURE 4. An instance of the visual of an Animation unit (ring)

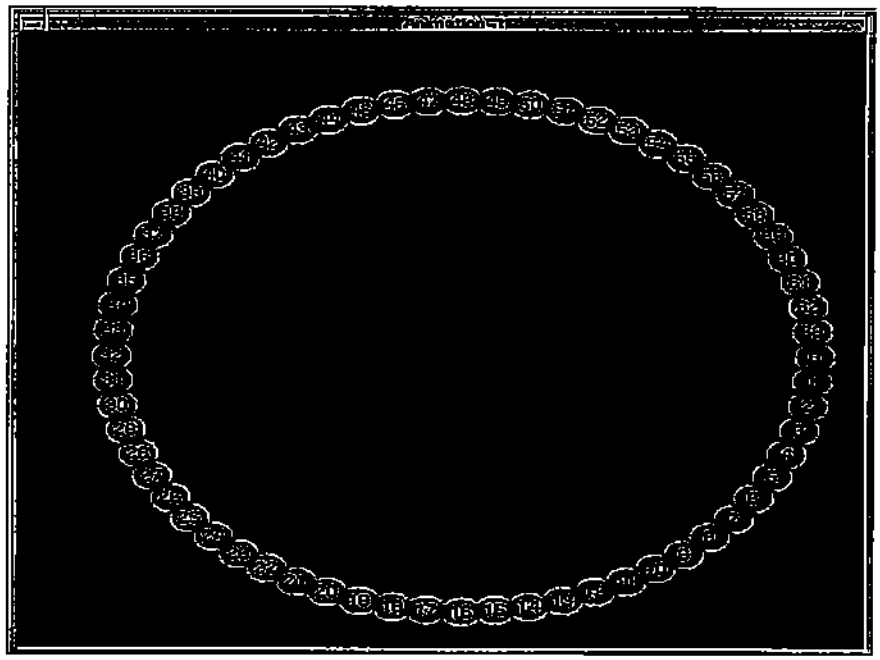


FIGURE 5. An instance of the visual of an Animation unit (array)

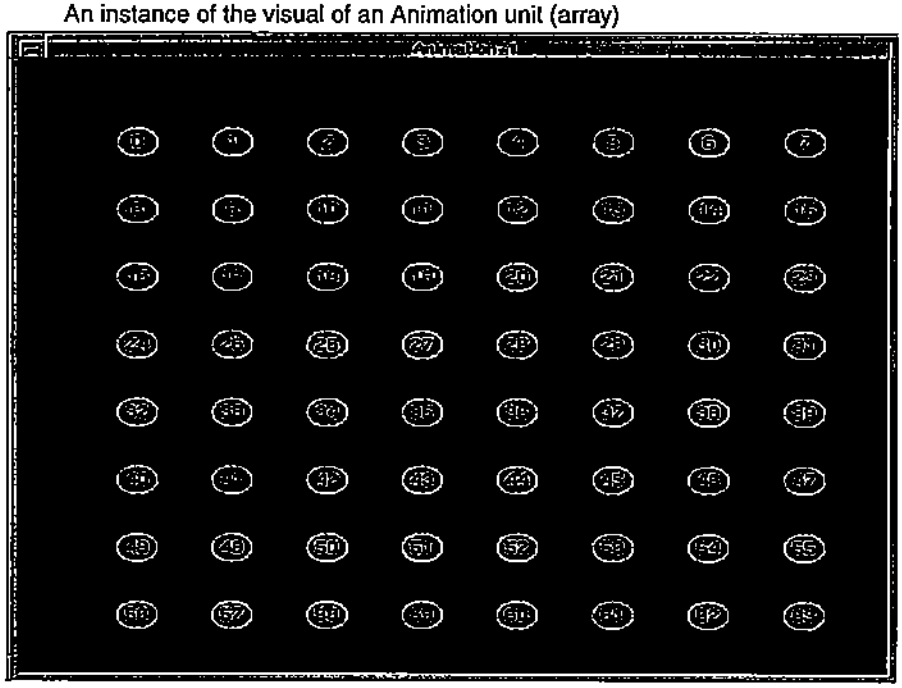


FIGURE 6. Binding the input to an Animation unit

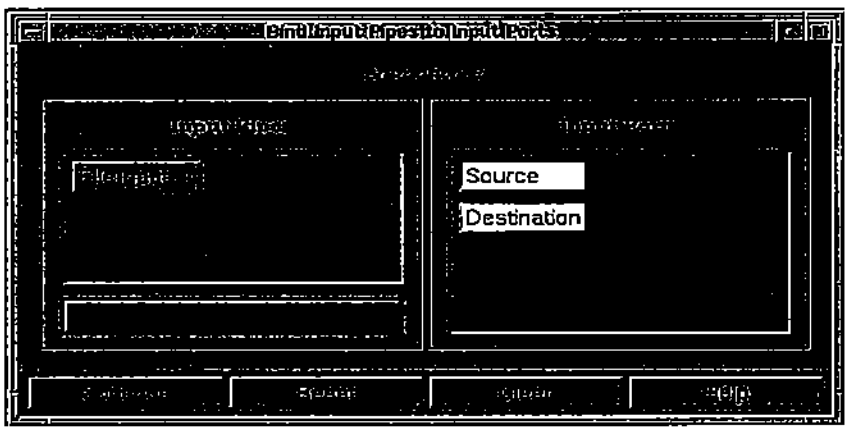


FIGURE 7. Binding the input to a filter-and-reduct unit

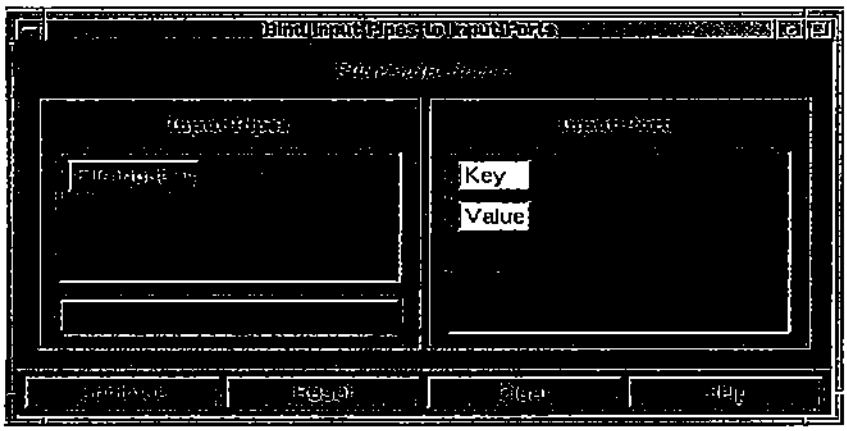
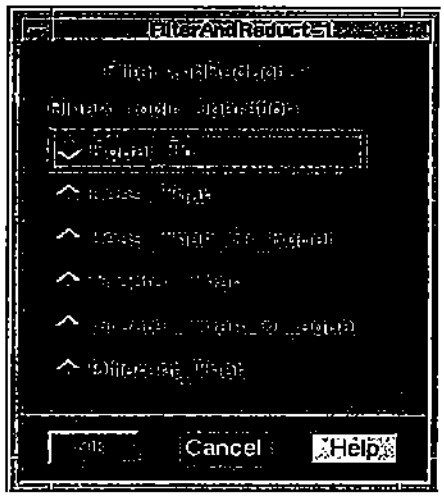


FIGURE 8. Configuring the filter-and-reduct unit



References

1. D.Allen, R.Bowker, K.Jourdenais, J.Simons, S.Sistare, R.Title, "The Prism Programming Environment", In Proceedings of Supercomputer Debugging Workshop '91, pp 1-7, Albuquerque, New Mexico, November 1991
2. R.A.Aydt, "SDDF: The Pablo Self-Describing Data Format", TR, UIUC, Dept. of CS, Sept. 1993
3. V.Balasundaram, G.Fox, K.Kennedy, U.Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions", Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPOPP), pp. 213-223, Williamsburg, Virginia, April 1991.
4. Beckman, Chatterjee, Cuny, Chapman, Chrisoides, Gannon, Grunwald, Hansen, Kesselman, Malony, Mehrota, Mohr, Sheffer, Sundaresan, Quinlan,Winnicka, Yang, Zima, "HPC++: A Draft White Paper", September 1995
5. T.Bemmerl, O.Hansen, T.Ludwig, "PATOP for Performance Tuning of Parallel Programs", in Proceedings of CONPAR 90, H.Burkhart (ed.) VAPP IV, Joint International Conference on Vector and Parallel Processing, vol 457 of Lecture Notes in Computer Science, pages 840-851, Zurich, Switzerland, Springer Verlag, Sept 1990
6. M.E.A.Berry, "The Perfect Club Benchmarks: Performance Evaluation of Supercomputers", The International Journal of Supercomputer Applications, 3, 3, pp. 5-40, Fall 1989
7. D.Breazeal, R.Anderson, W.D.Smith, W.Auld, K.Callghan, "A Parallel Software monitor for Debugging and Performance Tools on Distributed Memory Multicomputers", Proceedings of the Supercomputer Debugging Workshop, pp. 221-230, Dallas, October 1992
8. R.Butler, E.Lusk, "Monitors, Messages and Clusters: The P4 Parallel Programming System", Parallel Computing, 20, pp. 547-64, April 1994
9. E.Burke, "An Overview of System Software for the KSR 1", Digest of papers: CompCon Spring '93, pp. 295-299, San Francisco, IEEE Computer Society Press, February 1993
10. R.Calkin, R. Hempel, H. Hoppe, P. Wypior, "Portable Programming with the PARMACS Message Passing Library", Parallel Computing, 20, pp. 615-32, April 1994
11. D.Y.Cheng, "A Survey of Parallel Programming Tools", NAS Systems Division, TR RND-91-995, NASA Ames Research Center, Moffett Field, CA, May 1991
12. A.L.Couch, "Categories and Context in Scalable Execution Visualization", Journal of Parallel and Distributed Computing, 18, 2, June 1993, pp. 195-204
13. A.L.Couch, "Seecube User's Manual", TR, Tufts University, Dept. of CS, December 1987
14. A.L.Couch, D.W.Krumme, "Monitoring Parallel Executions in Real Time", Proceedings of the fifth Distributed Memory Computing Conference, pp. 1187-1206, April 1990
15. J.Dongarra, A.Geist, R.Manchek, V.Sunderam, "Integrated PVM Framework Supports Heterogeneous Network Computing", Computers in Physics, vol. 7, no. 2, pp. 166-75, April 1993

16. T.Fahringer, H.Zima, "A Static Parameter based Performance Prediction Tool for Parallel Programs", In Languages, Compilers, and Programming Environments for Parallel Systems - A Collection of Papers from the Vienna Group, vol TR 93-1, pp. 117-189, U. of Vienna, Institute for Software TEchnology and Parallel Systems, August 1993

17. J.Flower, A.Kolawa, "Programming with Express", Parasoft Corporation, RAPS Workshop, S.Augustin-Bonn, 1993

18. G.A.Geist, M.T.Heath, B.W.Peyton, P.H.Worley, "A User's Guide to PICL, a Portable Instrumentation Communication Library", TR ORNL/TM-11616, Oak Ridge National Laboratory, October 1990

19. S.Gillich, B.Ries, "Parallel Tool Environments - Infrastructure and Scalability Investigation", Esprit 6643, (Personal Communication), 1994

20. A.Gottlieb, K.Hwang, S.Sahni, "Special Issue on Tools and Methods for Visualization of Parallel Systems and Computations", vol 18, no. 2, Academic Press, June 1993

21. W.Gropp, E.Lusk, A.Skjellum, "Using MPI: Portable Parallel Programming with the Message Passing Interface", The MIT Press, 1994

22. S.Hackstadt, "Prototyping Advanced Parallel Program and Performance Visualizations", Master's Thesis, Dept. of CS, U. of Oregon, June 1994

23. S.Hackstadt, A.Malony, "Data Distribution Visualization (DDV) for Performance Evaluation", CIS-TR-93-21, Dept. of CS, U. of Oregon, June 1993

24. S.Hackstadt, A.Malony, "Visualizing Parallel Programs and Performance", IEEE Computer Graphics and Applications, Vol.15, No. 4, July 1995, pp. 12-14

25. S.Hackstadt, A.Malony, "Scalable Performance Visualization for Data-Parallel Programs", Proceedings of Scalable High Performance Computing Conference, IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp 342-349

26. M.Heath, A.Malony, D.Rover, "The Visual Display of Parallel Performance Data", to be published in Computer, Special Issue Parallel and Distributed Technology Tools, Nov. 1995

27. M.Heath, J.Etheridge, "Visualizing the Performance of Parallel Programs", IEEE Software, September 1991

28. M.T.Heath, J.A.Etheridge, "ParaGraph: A Tool for visualizing performance of parallel programs", U. of Illinois and ORNL, January 1992

29. V.Herrarte, E.Lusk, "Studying Parallel Program Behavior with Upshot", TR, ARNL, 1991

30. R.Hockney, "Performance Parameters and Benchmarking of Supercomputers", Parallel Computing, Vol. 17, no. 10-11, North Holland, December 1991

31. J. Hollingsworth, B.Irvin, B.P.Miller, "IPS User's Guide", TR, U. of Madison Wisconsin, Dept. of CS, October 1992

32. J.K.Hollingsworth, R.B.Irvin, B.P.Miller, "The Integration of Application and System Based Metrics in a Parallel Program Performance Tool", Proceedings of the third annual Symposium on PPOPP, SIGPLAN Notices, Vol. 26, no. 7, July 1991

33. HOOD Technical Group, "HOOD Reference Manual", July 1992

34. R.F. Boisvert, E.N. Houstis, J.R. Rice. "A system for Performance Evaluation of PDE software", IEEE Trans. Soft Eng, 19, pp. 418-425, 1979

35. IBM Corporation, "IBM Visualization Data Explorer User's Guide, 2nd Ed., August 1992

36. INTECS Sistemi, "HOODNice Reference Manual", Version 3.0, July 1993

37. C.H.Koelbel, D.B.Loveman, R.S.Schreiber, G.L.Steele, M.E.Zosel, "The High Performance Fortran Handbook", The MIT Press, 1994

38. D.Kileman, G.Sangudi, "Program Visualization by Integration of Advanced Compiler Technology with Configurable Views", TR, IBM T.J.Whatson Research Center, September 1992

39. E.Kraemer, J. Stasko, "The Visualization of Parallel Systems: An Overview", Journal of Parallel and Distributed Computing, 18, 1, June 1993, pp. 105-117

40. P. Kondapaneni, C.Pancake, C.Ward, "A Visual Programming Tool for Fortran D", December 1992

41. R.Koskela, M.Simmons (eds.), "Parallel Computer Systems: Performance Instrumentation and Visualization", Addison-Wesley, 1989

42. D.W.Krumme, A.L.Couch, B.R.House, J.Cox, "The Triplex Tool Set for the NCUBE Multiprocessor", TR, Tufts University, Dept. of CS, June 1989

43. T.LeBlanc, J.Mellor-Crummey, R.Cheng, "Data Parallel Program Visualizations from Formal Specifications", Journal of Parallel and Distributed Computing, 9, 2, June 1990, pp. 203-217

44. A.Malony, D.Reed, "Visualizing Parallel Computer Performance" in Instrumentation for Future Parallel Computer Systems, M.Simmons, R.Koskela, I.Bucher (eds), ACM Press, New York, NY, 1989, pp. 59-90

45. A.Malony, B.Mohr, P.Beckman, D. Gannon, S.Yang, F.Bodin, "Performance Analysis of pC++: A Portable Data Parallel Programming System for Scalable Parallel Computers", Proc. of the International Parallel Processing Symposium, April 1994

46. A.Malony, D.A.Reed, H.A.G.Wijhoff, "Performance Measurement Intrusion and Perturbation Analysis", IEEE Trans. on Parallel and Distributed Systems, Vol. 3, no. 4, July 1992

47. D.C.Marinescu et. al., "Models for Monitoring and Debugging Tools for Parallel and Distributed Software", Journal of Parallel and Distributed Computing, September 1990

48. B.Miller, "What to Draw? When to Draw? An Essay on Parallel Program Visualization", Journal of Parallel and Distributed Computing, 18, 2, June 1993, pp. 265-269

49. D.A.Reed, "Performance Instrumentation Techniques for Parallel Systems", in Models and Techniques for Performance Evaluation of Computer and Communication Systems, L.Donatiello, R.Nelson (eds.) Springer Verlag Lecture Notes in Computer Science, 1993

50. D.A.Reed, R.A.Aydt, T.M.Madhyasta, R.J.Noe, K.A.Shields, B.W.Schwartz, "An Overview of the Pablo Performance Analysis Environment", TR, UIUC, Dept. of CS, Sept. 1993

51. D.A.Reed, R.A.Aydt, R.J.Noe, P.C.Roth, K.A.Shields, B.W.Schwartz, L.F.Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment", In Proceedings of the Scalable Parallel Libraries Conference, A.Skjellum (ed.), IEEE Computer Society, 1993

52. D.A.Reed, "Experimental Analysis of Parallel Systems: Techniques and Open Problems", TR, UIUC, Dept. of CS, 1994

53. D. Rover, "A Performance Visualization Paradigm for Data Parallel Computing", Proc. of the 25th Hawaii International Conference on System Sciences, 1992

54. D.Rover, C. Wright, "Visualizing the Performance of SPMD and Data Parallel Programs", Journal of Parallel and Distributed Computing, 18, 2, June 1993, pp. 129-146

55. S.Sarukkai, D.Gannon, "Parallel Program Visualization using SIEVE.1", Proceedings of the 1992 ACM International Conference on Supercomputing, Washington D.C., July 1992

56. S.Sarukkai, D.Gannon, "SIEVE: A Performance Debugging Environment for Parallel Programs", Journal of Parallel and Distributed Computing, 18, 2, June 1993, pp. 147-168

57. S.Sistare, D.Allen, R.Bowker, K.Jourdenais, J.Simons, R.Title, "Data Visualization and Performance Analysis in the Prism Programming Environment", In N.Tophan, R.Ibbett, T.Bemmerl, (eds.) Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing, volume A-11 of IFIP transactions, pp. 37-52, Edinburgh, North-Holland, April 1992

58. S.Srinivas, D.Gannon, "Interactive Visualization and Animation of Parallel Programs", TR, Indiana University

59. J.Stasko, E.Kraemer, "A Methodology for Building Application-specific Visualizations of Parallel Programs", Journal of Parallel and Distributed Computing, 18, 1, June 1993, pp. 258-264

60. L.H.Tourcotte, "A Survey of Software Environments for Exploiting Networked Computing Resources", TR, Engineering Research Center for Computational Field Simulation, June 1993

61. E.Tufte, "Envisioning Information", Graphics Press, CT, April 1991

62. W.Williams, T.Hoel, D.Pase, "The MPP Apprentice", conference on programming environments for massively parallel computers, Ascona, Switzerland, April 1994

63. J.C.Jan, "Performance Tuning with AIMS - An Automated Instrumentation and Monitoring System for Multicomputers", Proceedings of the 27th Hawaii International Conference on System Sciences, ACM, January 1994

Appendix

In this appendix we present a closer look on the design of the Patool. The Patool has been designed using the Hoodnice [36] and the actual design data are maintained with the tool. The Hoodnice is a software engineering package that manages the complete product cycle. In the subsequent chapters we give an edited summary of the objects making up the Patool following the format suggested by the HOOD [33] methodology.

Each section gives a brief description of the problem to be solved by the object, a brief description of the strategy to follow, a summary of each methods and a HOOD graphical description. For more detail the Hoodnice tool must be used.

The last section of the appendix is a summary listing of the source files making up this object. These source files refer to the Patool source code tree.

6.0 Patool

6.1 Problem Definition

The performance analysis and visualization tool (**Patool**) is presented with the problem of analyzing and presenting trace data from a parallel application run, in an intuitive, graphical manner. The Patool bases its operation on the availability of trace data prior to its invocation. The data have been collected elsewhere in the system in a file under some format. Because of the importance of the format the data are stored in, we have selected to adopt the SDDF. Under this format, the structure and semantics of the data need not be hardwired in the Patool. They are rather included in the trace data themselves. The other important problem the Patool targets is the necessity for different analysis layouts, based on the specific problem being analyzed. To overcome this problem the Patool incorporates the notion of the Analysis Graph. This graph is an ordinary Directed Acyclic Graph (DAG) where the nodes are functional units and the edges denote the flow of information (data) among the functional units.

Because a number of these ideas appear in existing research tools, especially Pablo, Patool is making use of the available knowledge and builds on top of it.

The Patool design is totally object oriented and based on the notion of an abstract module that provides a set of services to its environment.

In the subsequent sections a complete analysis of the components of the Patool is presented.

6.2 Formalization of the strategy

The Patool is structured in a number of modules which provide a set of different high level functionality. At this level, the necessary components are identified as the main interface of the Patool, the accessing (of the data), the set of all functional units which

are available to the system, the set of filtering and reduction techniques as well as the graphical possibilities for the definition of an analysis graph, the management and, of course, the trace data specifications. Each one of these components identifies a different set of functionalities, which when brought together, built up the Patool.

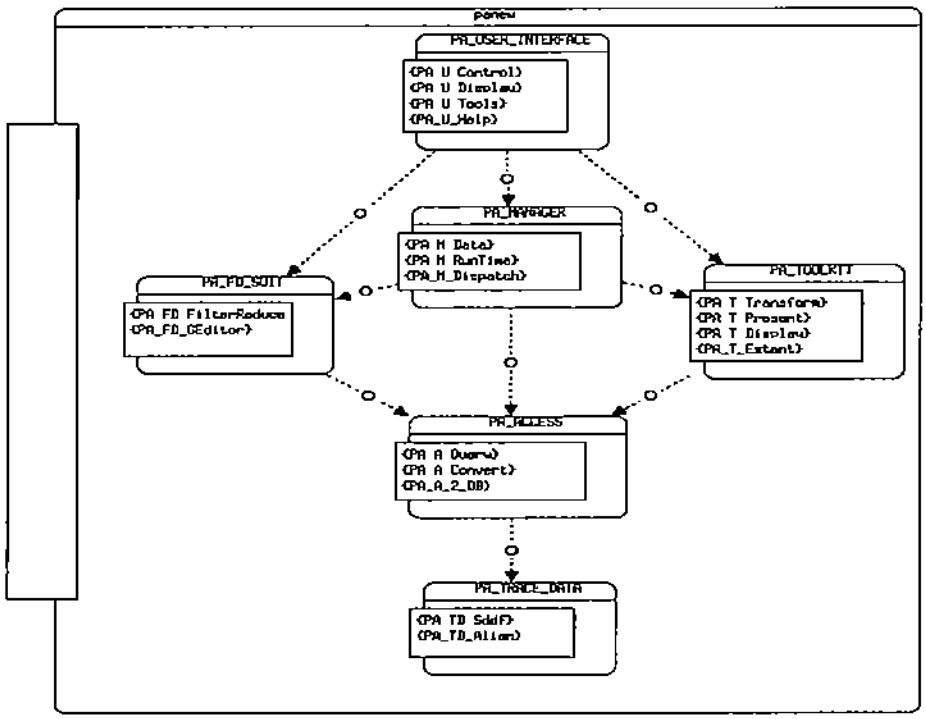
Identification of the Objects and Operations

TABLE 18. Patool Objects and Operations

Object Name	Object Type	Object Operations
Pa_User_Interface	Passive	Pa_U_Control
		Pa_U_Display
		Pa_U_Tools
		Pa_U_Help
Pa_Manager	Passive	Pa_M_Data
		Pa_M_RunTime
		Pa_M_Dispatch
Pa_FdSuit	Passive	Pa_Fd_FilterReduce
		Pa_Fd_Geditor
Pa_Toolkit	Passive	Pa_T_Transform
		Pa_T_Present
		Pa_T_Display
		Pa_T_Extent
Pa_Access	Passive	Pa_A_Query
		Pa_A_Convert
		Pa_A_2Sddf
Pa_Trace_Data	Passive	Pa_Td_Sddf
		Pa_Td_Alien

Graphical Description

FIGURE 9. Patool Object Hood Graphical Description



Justification of the Design Decisions

We have chosen to separate the functionality of the various modules in a way that enables reusability, maintenance and portability. Although the format of our choice is Sddf we do not exclude the possibility for a "plug and go" solution of a new format, referred to as "alien" in this context. We simply require that its specification is plugged in the appropriate module.

We have identified and separated all the functionality of interacting and communicating data within the system in a separate module. This provides the grounds for future optimization of the functionality and the easy maintenance in case of major structural changes.

The notion of a functional unit is by itself an independent object which specifies the input it expects and produces a well defined output. We have separated the functionality of a functional unit in a separate module which constitutes a library of available analysis units. This library is possible to be expanded in an easy and straightforward manner, by simply plugging a new functional unit.

7.0 Pa_UserInterface

7.1 Problem Definition

The abstraction provided by the user interface module serves as the front-end of the whole Patool. The functionality is divided in four submodules with different responsibilities. The *Control* provides all the basic functions such as file operations, as well as an unspecified portion which has to do with the specific controls of the system, such as "run", "stop", "step", "reset" and so on. The *Tools* submodule contains all the user interface commands with the graphics editor for specifying the computation and transformation graph, as well as commands that can be used for extension, such as connection with other tools that could extend the overall Patool functionality. The *Display* submodule contains all the functionality needed for manipulating the various graphic operations and displays that will usually appear in performance analysis and visualization session. Finally, the *Help* submodule contains basic operations for assisting the user in best exploiting and using the Patool.

7.2 Formalization of the strategy

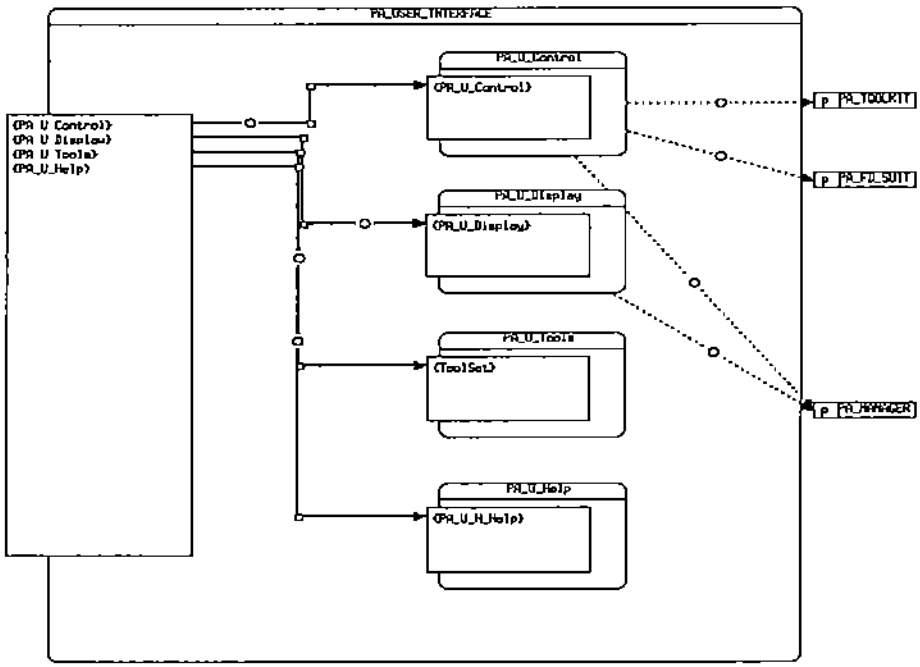
Identification of the Objects and Operations

TABLE 19. User Interface Objects and Operations

Object Name	Object Type	Object Operations
Pa_U_Control	Passive	Pa_U_C_Patool Pa_U_C_menuOps Pa_U_C_resourceManagOps
Pa_U_Display	Passive	Pa_U_D_defaultsDialog Pa_U_D_FileSelectionDialog Pa_U_D_GeneralDialogBox
Pa_U_Help	Passive	Pa_U_H_Help
Pa_U_Tools	Passive	Pa_U_T_toolset Pa_U_T_connectOther

Graphical Description

FIGURE 10. User Interface Object Hood Graphical Description



8.0 Pa_U_Control

8.1 Problem Definition

The **Control** module provides all the necessary sets of operations for managing the tool resources, creating the tool menus, and, of course, providing the kernel of the system (in cooperation with the manager module).

8.2 Formalization of the strategy

The Control module is implemented through three different submodules. The main control object is the one that brings up the whole system by performing in an iterative fashion the initialization of the system and by activating the kernel. The menus object provide the envelope for initializing and binding operations to the various menus for controlling the environment. The tool resources object provides the interface for setting up the resources of the vast number of graphical objects comprising the system. The

interface provides a default setting for all resource attributes pertaining to the windowing environment.

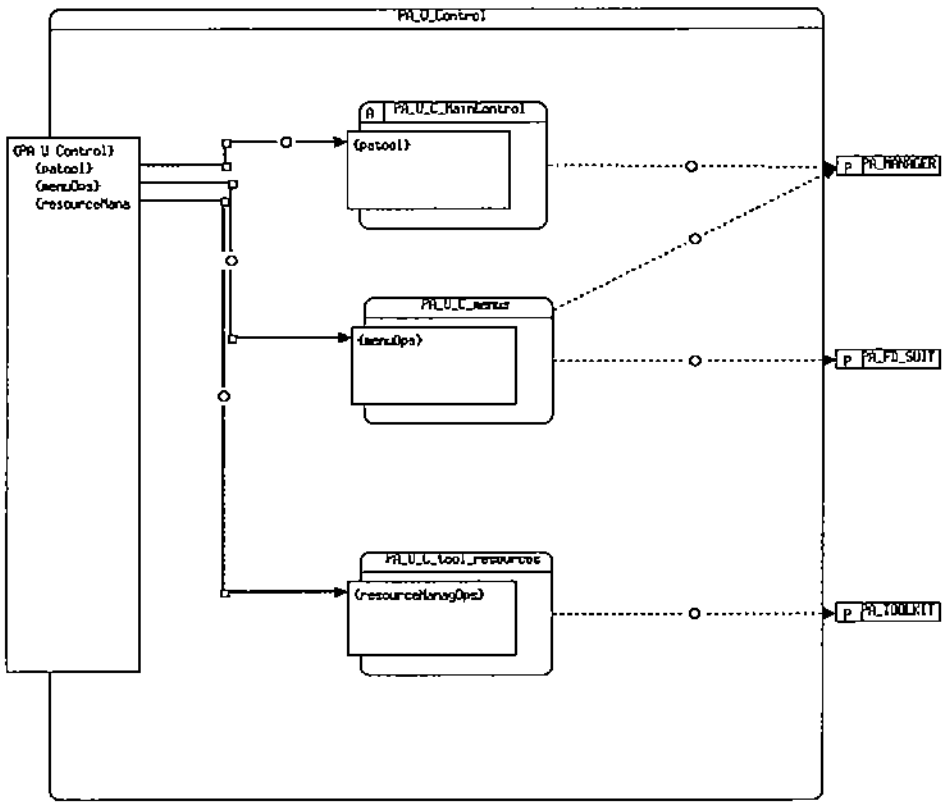
Identification of the Objects and Operations

TABLE 20. Control Objects and Operations

Object Name	Object Type	Object Operations
Pa_U_C_MainControl	Active	Pa_U_C_delete Pa_U_C_create Pa_U_C_patool
Pa_U_C_tool_resources	Passive	Pa_U_C_resourceManagOps
Pa_U_C_menus	Passive	Pa_U_C_menuOps

Graphical Description

FIGURE 11. Control Object Hood Graphical Description



9.0 Pa_U_Display

9.1 Problem Definition

The **Display** module contains all the functionality that has to do with the interaction of the Patool with the user. It provides the necessary dialog services for importing and exporting information to the system

9.2 Formalization of the strategy

The Display module consists of three basic modules which incorporate various dialog services. The Defaults dialog is an object that provides a set of operations for instantiat-

ing an interactive session with the user that will specify default settings for the various functional units.

The File selection dialog is an object that provides interface to the classical file selection dialog. This dialog is basically used when the user requires to load or store some portion of the configuration information in a file, or when the trace file to be used needs to be specified.

The General dialog is used to provide assistance whenever a communication of information to the user is required. It provides fixed operations for error reporting and information presentation. It is further customized for specific dialog needs.

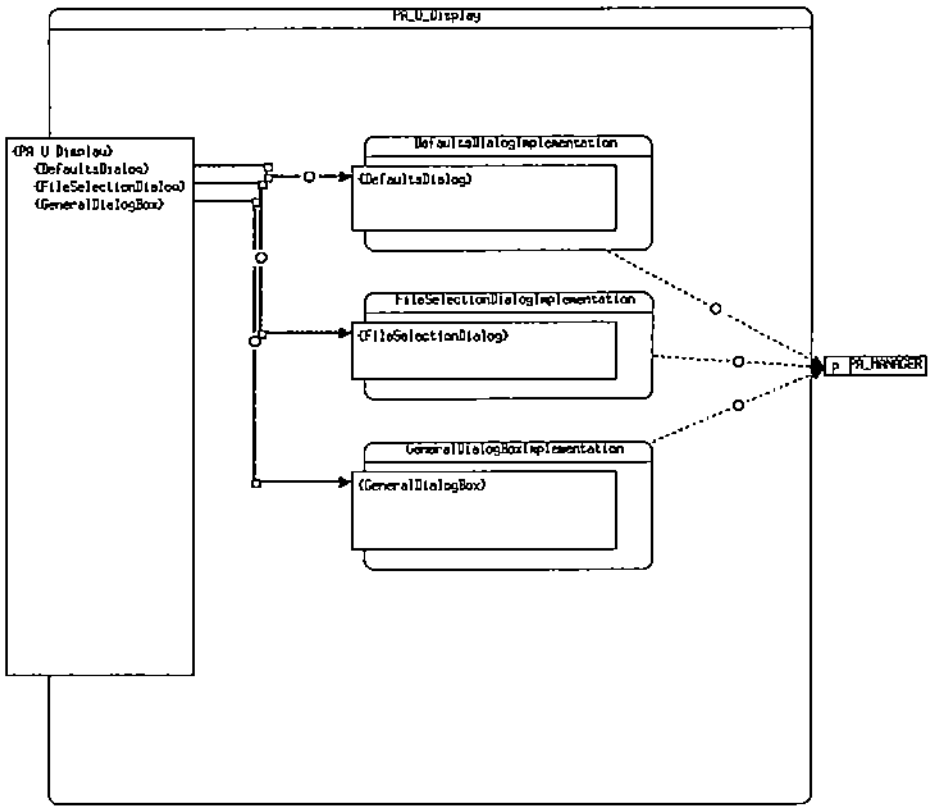
Identification of the Objects and Operations

TABLE 21. Display Objects and Operations

Object Name	Object Type	Object Operations
Pa_U_D_GeneralDialogBox	Passive	Pa_U_D_G_RunWarning
		Pa_U_D_G_RunInfo
		Pa_U_D_G_RunError
		Pa_U_D_G_Run
		Pa_U_D_G_Destroy
		Pa_U_D_G_Create
Pa_U_D_FileSelectionDialog	Passive	Pa_U_D_F_GetFileName
		Pa_U_D_F_Run
		Pa_U_D_F_Destroy
		Pa_U_D_F_Create
Pa_U_D_DefaultsDialog	Passive	Pa_U_D_D_Raise
		Pa_U_D_D_Destroy
		Pa_U_D_D_Create

Graphical Description

FIGURE 12. Display Object Hood Graphical Description



10.0 Pa_U_Help

10.1 Problem Definition

The **Help** module provides the functionality of help to the system. The Help is built in a hierarchical directory fashion with hyperlinks and cross references. It can be activated by topic or by context.

10.2 Formalization of the strategy

The source text of the help is constructed and stored statically in a structure resembling the nesting of tool operations. During the execution the various help topics become available through interfacing with the help object, which has a set of operations to manage the help system.

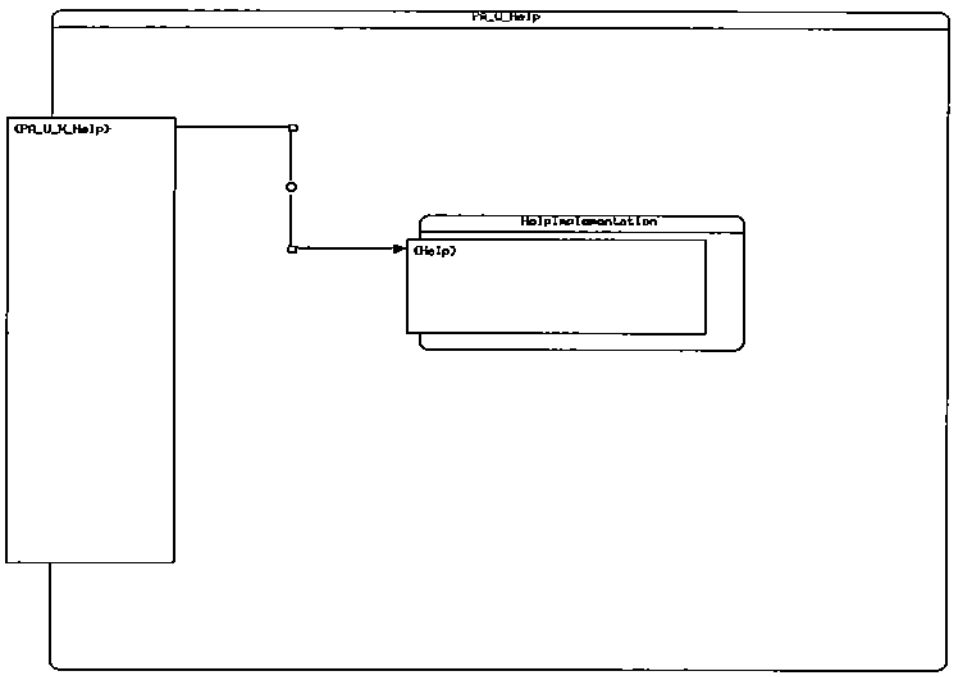
Identification of the Objects and Operations

TABLE 22. Help Objects and Operations

Object Name	Object Type	Object Operations
Pa_U_H_Implementation	Passive	Pa_U_H_SetHelpSource
		Pa_U_H_GiveOn
		Pa_U_H_Destroy
		Pa_U_H_Create

Graphical Description

FIGURE 13. Help Object Hood Graphical Description



11.0 Pa_U_Tools

11.1 Problem Definition

The **Tools** module provides the necessary functionality to connect to alien tools (external tools) such as the graphical analysis editor. The tools that are hooked via this inter-

face include tools that could be run independently of the Patool but also those portions of the Patool that could be run independently, such as the converters and the graphical editor.

11.2 Formalization of the strategy

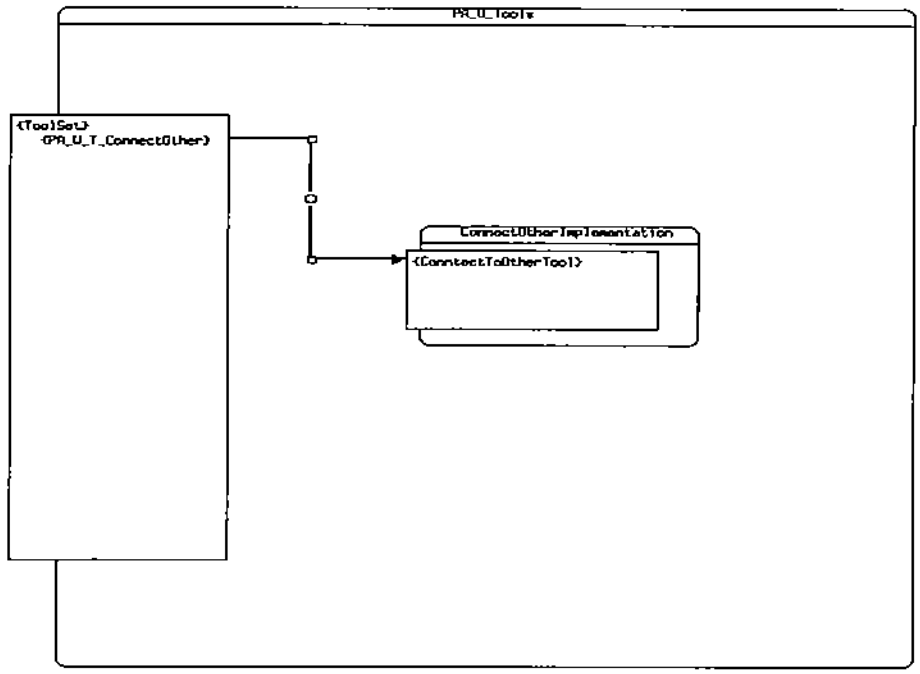
Identification of the Objects and Operations

TABLE 23. Tools Objects and Operations

Object Name	Object Type	Object Operations
Pa_U_T_ConnectOther	Passive	Pa_U_H_SetHelpSource
		Pa_U_H_GiveOn
		Pa_U_H_Destroy
		Pa_U_H_Create

Graphical Description

FIGURE 14. Tools Object Hood Graphical Description



12.0 Pa_Manager

12.1 Problem Definition

The **Manager** module deals with many of the issues of running the Patool and provides the appropriate functionality. It is developed in three different modules, each with a predefined non overlapping functionality.

The Data submodule contains all the necessary functionality for ensuring the flow of trace data around the system. It is implemented via the Pa_Access provided services.

The Run Time module provides the actual running kernel (instance) of the Patool, generating and managing all the provided services.

The Dispatch module contains all the functionality needed for providing the necessary triggering for the analysis graph execution and data packet handling. It also serves as the manager of things, keeping an eye around the system for inconsistencies and potential problems.

12.2 Formalization of the strategy

The Manager module as mentioned earlier provides the infrastructure support for the overall tool. We have separated the infrastructure in two main areas. That of taking care of data movements, i.e. providing the connection management among the functional units, ensuring the flow and validity of the data, and that of taking care of the control and sequence of the operations. The first is provided by the data object and the latter by the dispatcher object. The run time kernel is making use of both these services in order to realize the system.

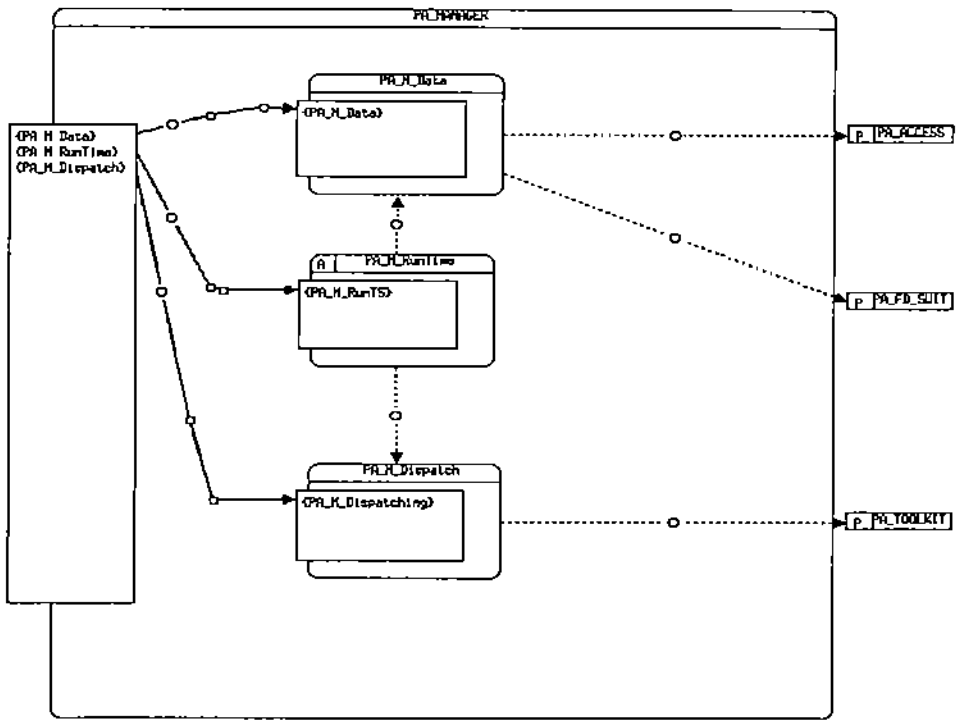
Identification of the Objects and Operations

TABLE 24. Manager Objects and Operations

Object Name	Object Type	Object Operations
Pa_M_RunTime	Active	Pa_M_RunTS
Pa_M_Dispatch	Passive	Pa_M_Dispatching
Pa_M_Data	Passive	Pa_M_Data

Graphical Description

FIGURE 15. Manager Object Hood Graphical Description



13.0 Pa_M_RunTime

13.1 Problem Definition

The **Run Time** module provides the running kernel of Patool instance.

13.2 Formalization of the strategy

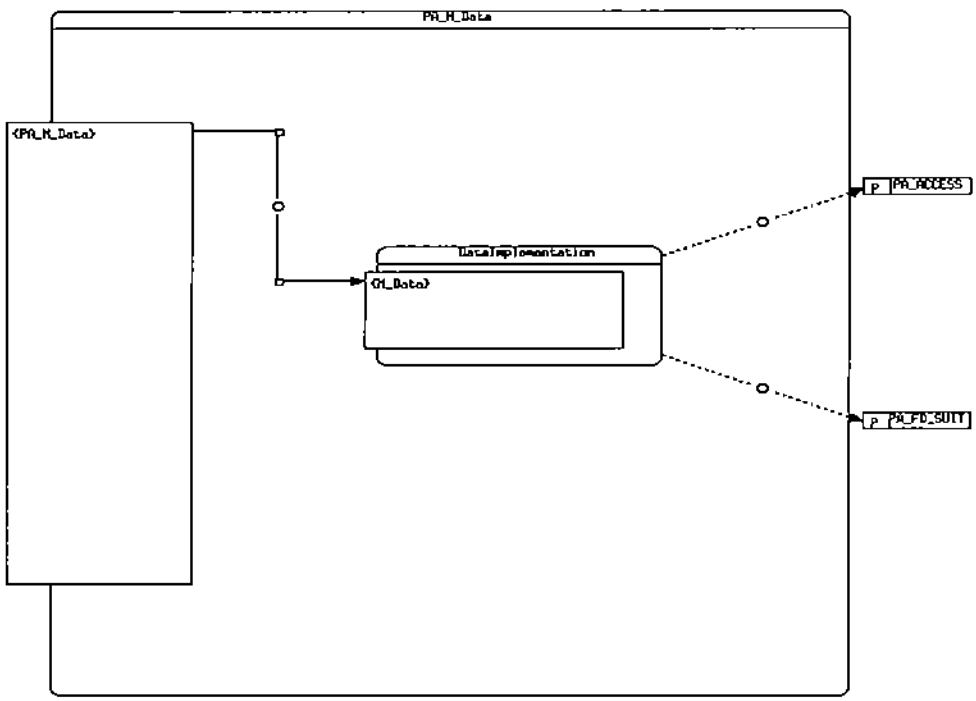
Identification of the Objects and Operations

TABLE 25. Pa_M_RunTime Objects and Operations

Object Name	Object Type	Object Operations
Pa_M_RT_Implementation	Active	Pa_M_RT_Run
		Pa_M_RT_ReadAutomaton
		Pa_M_RT_CreateMenu
		Pa_M_RT_CreateGUI

Graphical Description

FIGURE 16. Pa_M_RunTime Object Hood Graphical Description



14.0 Pa_M_Dispatch

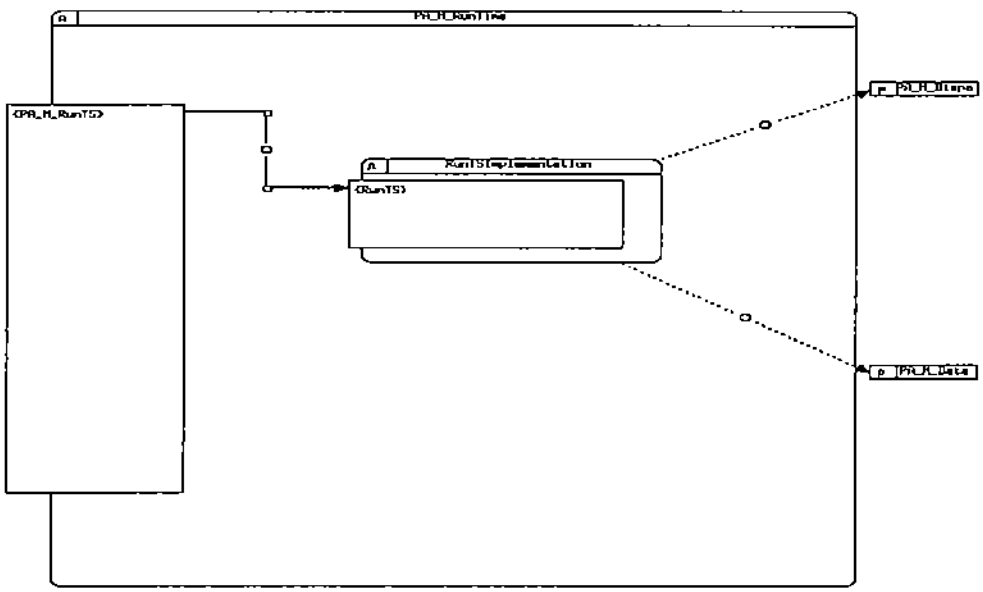
Identification of the Objects and Operations

TABLE 26. Pa_M_Dispatch Objects and Operations

Object Name	Object Type	Object Operations
Pa_M_D_Implementation	Passive	Pa_M_D_SavetoFile
		Pa_M_D_LoadfromFile
		Pa_M_D_ResetGraph
		Pa_M_D_Destroy
		Pa_M_D_Create
		Pa_M_D_RunGraph
		Pa_M_D_SetUpGraph

Graphical Description

FIGURE 17. Pa_M_Dispatch Object Hood Graphical Description



15.0 Pa_M_Data

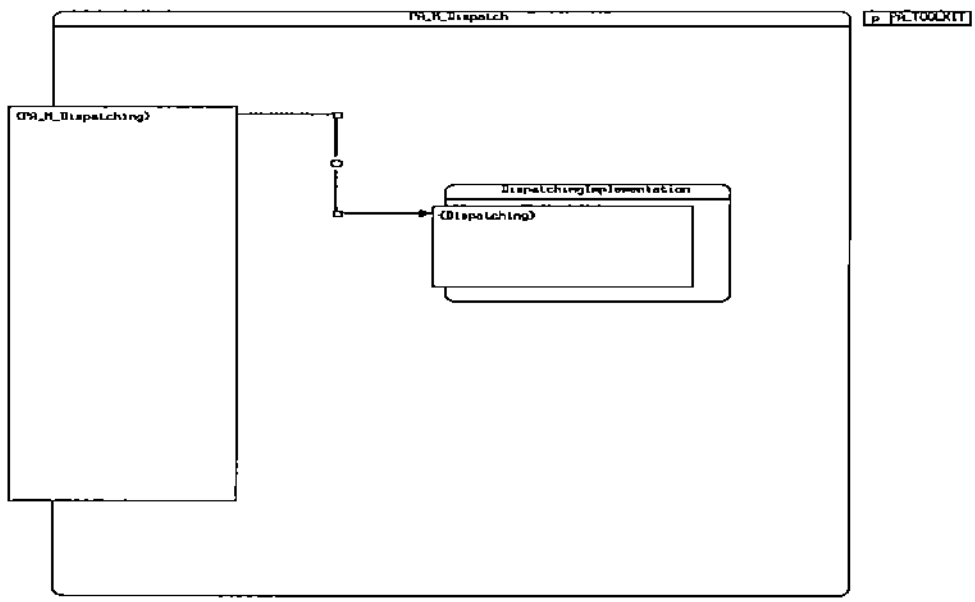
Identification of the Objects and Operations

TABLE 27. Pa_M_Data Objects and Operations

Object Name	Object Type	Object Operations
Pa_M_Da_Implementation	Passive	Pa_M_Da_Destroy
		Pa_M_Da_SaveOut
		Pa_M_Da_Create
		Pa_M_Da_LoadInGraph
		Pa_M_Da_Locate

Graphical Description

FIGURE 18. Pa_M_Data Object Hood Graphical Description



16.0 Pa_FdSuit

16.1 Problem Definition

The **FdSuit** module encloses the core of the functions needed to treat data for large size parallel problems. It provides an interface towards the scalability of the various views and displays. For small size problems it can be bypassed, since there is no need to use such techniques. It is provided as a separate module so as to allow for extensibility and transferability. Also the DAG graphical editor is included.

16.2 Formalization of the strategy

The functionality of filtering and reduction has been realized through two different approaches. The first is the definition of a set of operations in the functional unit style that allow the flow of data to be selectively communicated in the attached analysis graph nodes. The second, is the functionality of the DAG graphical editor itself, since it is the basic medium to describe the flow of data and the various reductions that need to be done, from a level higher than in the functional unit. The DAG provides a high level description of the intended analysis. As already mentioned it does so by providing two basic notions. That of functional unit, represented by a node, and that of the data flow between the functional units represented by the graph edges.

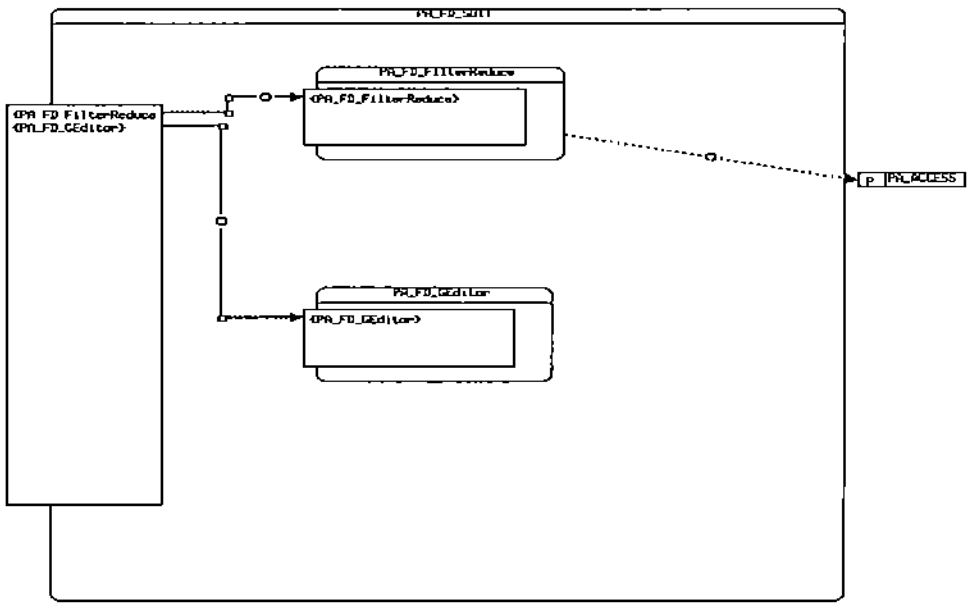
Identification of the Objects and Operations

TABLE 28. Pa_FdSuit Objects and Operations

Object Name	Object Type	Object Operations
Pa_Fd_FilterReduce	Passive	Pa_Fd_filterReduce
Pa_Fd_Geditor	Passive	Pa_Fd_Geditor
		Pa_Fd_displayOps
		Pa_Fd_nodeOps
		Pa_Fd_edgeOps

Graphical Description

FIGURE 19. Pa_FdSuit Object Hood Graphical Description



17.0 Pa_Fd_FilterReduce

17.1 Problem Definition

The filtering and reduction module encloses the functionality for filtering and reducing the data using either a predefined algorithm or in an interactive manner. The functionality of the filtering and reduction can be either used as a stand-alone tool, or in conjunction with an analysis graph to alter the normal flow.

In the context of a DAG analysis graph, the functionality is delivered via a functional unit. In a stand-alone version, the functionality can be used to minimize or to preprocess the contents of a specified trace file.

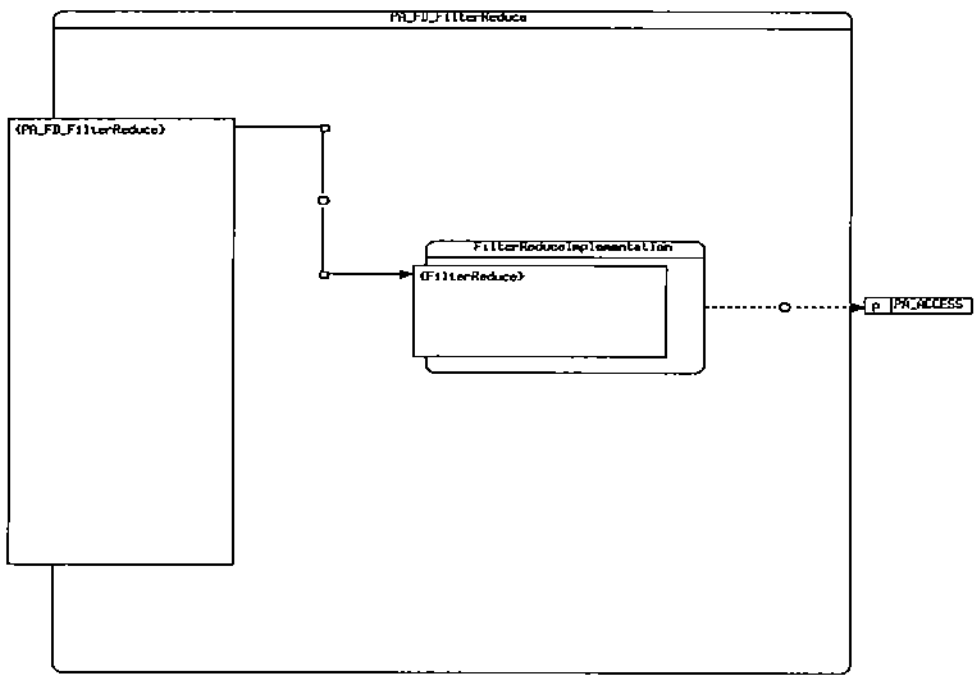
Identification of the Objects and Operations

TABLE 29. Pa_Fd_FilterReduce Objects and Operations

Object Name	Object Type	Object Operations
Pa_Fd_Implementation	Passive	Pa_Fd_Apply
		Pa_Fd_Query
		Pa_Fd_Configure
		Pa_Fd_Init
		Pa_Fd_Destroy
		Pa_Fd_Create
		Pa_Fd_Reduce

Graphical Description

FIGURE 20. Pa_Fd_FilterReduce Object Hood Graphical Description



18.0 Pa_Fd_GEditor

18.1 Problem Definition

The **Graphical Editor** module provides all necessary functionality for the creation and management of the DAG analysis infrastructure.

18.2 Formalization of the strategy

The notion of a DAG is decomposed in edges and nodes along with a set of displaying management routines.

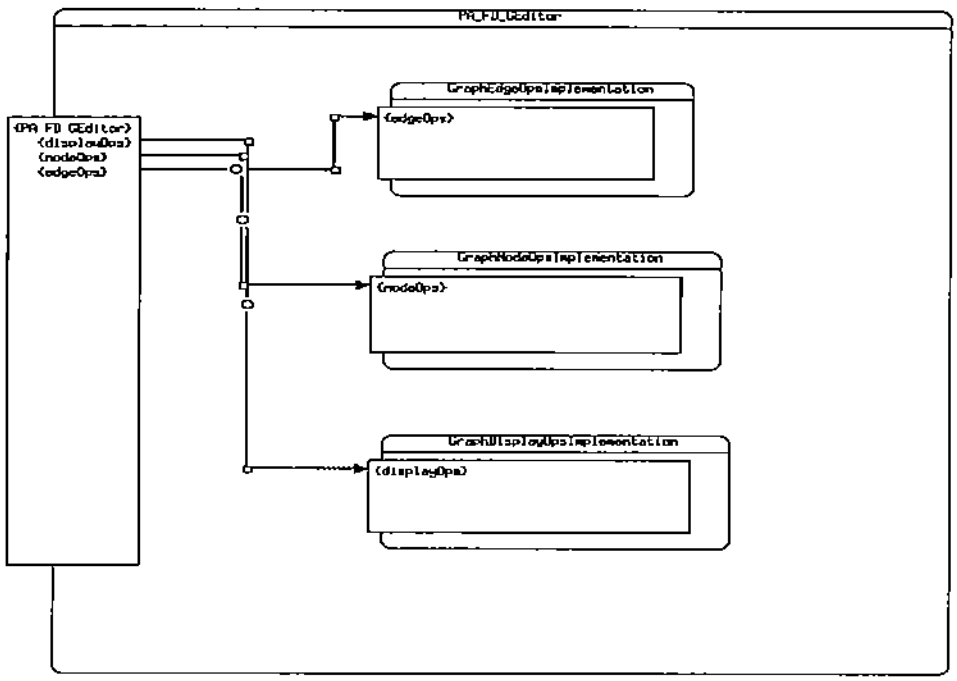
Identification of the Objects and Operations

TABLE 30. Pa_Fd_GEditor Objects and Operations

Object Name	Object Type	Object Operations
Pa_Fd_Ge_EdgeOps	Passive	Pa_Fd_Ge_E_setEnd
		Pa_Fd_Ge_E_setStart
		Pa_Fd_Ge_E_setSourceNode
		Pa_Fd_Ge_E_Destroy
		Pa_Fd_Ge_E_Create
Pa_Fd_Ge_NodeOps	Passive	Pa_Fd_Ge_N_EventHandler
		Pa_Fd_Ge_N_moveTo
		Pa_Fd_Ge_N_getTopCenter
		Pa_Fd_Ge_N_NodeName
		Pa_Fd_Ge_N_getBottomCenter
		Pa_Fd_Ge_N_addOutEdge
		Pa_Fd_Ge_N_addInEdge
		Pa_Fd_Ge_N_Destroy
		Pa_Fd_Ge_N_Create
Pa_Fd_Ge_GraphDisplay	Passive	Pa_Fd_Ge_G_writeLayouttoFP
		Pa_Fd_Ge_G_Reset
		Pa_Fd_Ge_G_NodeCount
		Pa_Fd_Ge_G_getNodeIndex
		Pa_Fd_Ge_G_DeleteNode
		Pa_Fd_Ge_G_containsNode
		Pa_Fd_Ge_G_AddNode
		Pa_Fd_Ge_G_AddEdge

Graphical Description

FIGURE 21. Pa_Fd_GEditor Object Hood Graphical Description



19.0 Pa_Toolkit

19.1 Problem Definition

The **Toolkit** module encloses all the functionality that has to do with the actual data manipulation, calculation, analysis and processing. It provides a set of modules and functions that assist in implementing a performance analysis session, including presentation modules, displaying techniques and data transformation. The various submodules are often identified as functional units.

There is also provision for an extension interface that serves the purpose of the extensibility of the tool by providing an interface abstraction. The motivation of keeping the various functionalities in this object separate results from the need to have core analysis functions as independent and as portable as possible. In principle they could be taken out and plugged to some other environment, provided the interface requirements remain unchanged.

19.2 Formalization of the strategy

The Toolkit is separated in four main modules. The Extent module provides a template for functional unit extension, and is made available for maintenance and extension reasons. The Transform module provides a number of modules which are referred to as transforming functional units. Usually these units serve as internal nodes in the analysis graph, transforming their input to their output based on some functionality. The Presentation module provides a number of modules which are referred to as presentation functional units. They are usually terminal nodes in the analysis graph and present in a graphical fashion the data coming into their input. The Display module provides a number of modules which again serve as terminal nodes in the analysis graph and which provide aggregate graphical displays of their input.

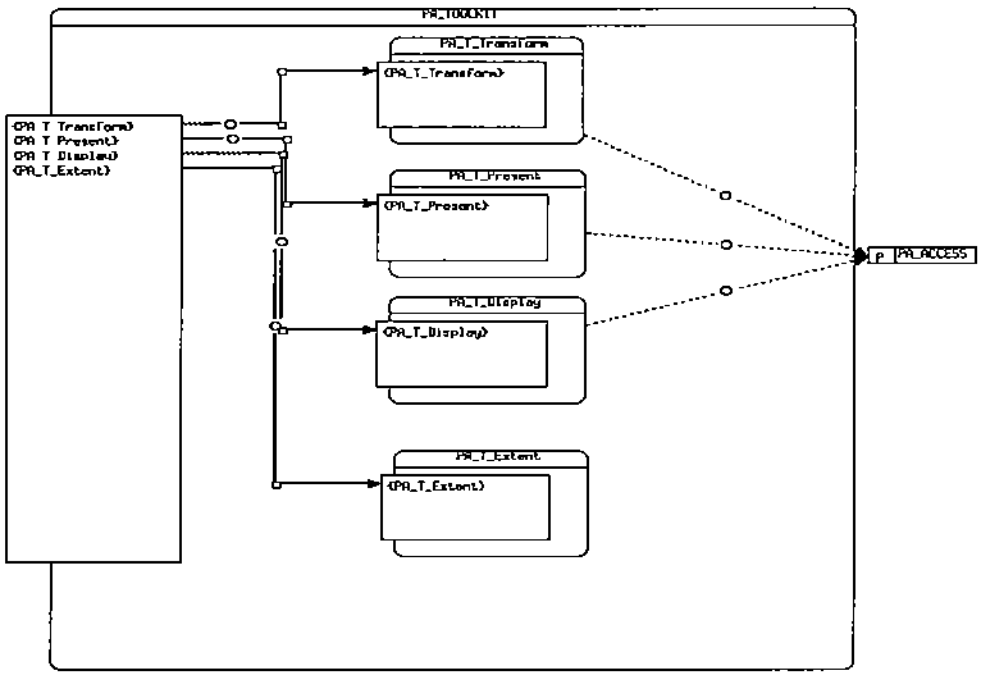
Identification of the Objects and Operations

TABLE 31. Pa_Toolkit Objects and Operations

Object Name	Object Type	Object Operations
Pa_T_Transform	Passive	Pa_T_T_Scale
		Pa_T_T_ReductionMath
		Pa_T_T_Logarithm
		Pa_T_T_Power
		Pa_T_T_Unary
		Pa_T_T_Binary
		Pa_T_T_Trigonometry
		Pa_T_T_SynthCoord
		Pa_T_T_SynthVector
		Pa_T_T_SynthArray
Pa_T_Present	Passive	Pa_T_P_PolarPlot
		Pa_T_P_BarGraph
		Pa_T_P_Bubble
		Pa_T_P_Chart
		Pa_T_P_Cluster
		Pa_T_P_Contour
		Pa_T_P_Dial
		Pa_T_P_Interval
		Pa_T_P_Kiviat
		Pa_T_P_Matrix
		Pa_T_P_Piechart
		Pa_T_P_Scatter3D
		Pa_T_P_XYGraph
		Pa_T_P_Led
		Pa_T_P_HistoryDial
Pa_T_Display	Passive	Pa_T_D_Animation
		Pa_T_D_Zooming

Graphical Description

FIGURE 22. Pa_Toolkit Object Hood Graphical Description



20.0 Pa_T_Transform

20.1 Problem Definition

The Transform module offers a set of operations that manipulate the stream of trace data to produce a new set that has undergone some sort of transformation. The actual functionality of transformation is implemented via the lower level module, depending on the type of transformation. The collection of functional units includes a number of mathematical and structural transformations.

20.2 Formalization of the strategy

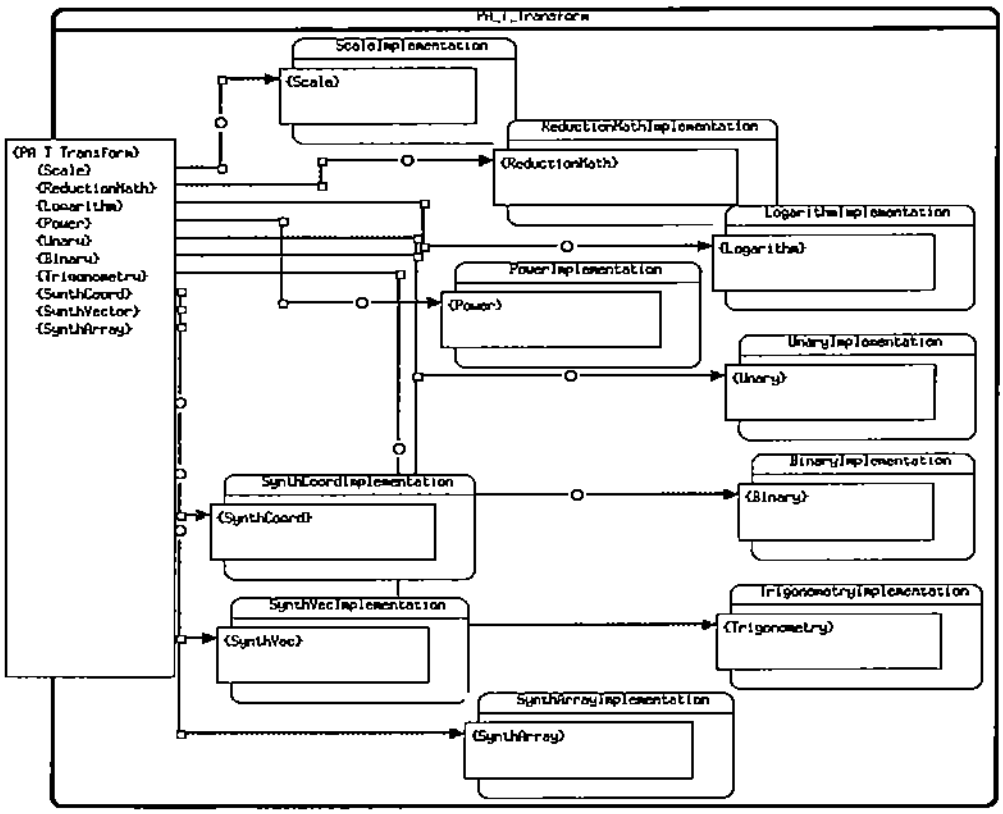
Identification of the Objects and Operations

TABLE 32. Pa_T_Transform Objects and Operations

Object Name	Object Type	Object Operations
Pa_T_T_SynthCoord	Passive	Pa_T_T_SC_Run Pa_T_T_SC_Ready Pa_T_T_SC_SaveToFile Pa_T_T_SC_LoadFromFile Pa_T_T_SC_Init Pa_T_T_SC_Copy Pa_T_T_SC_Configure
Pa_T_T_SynthVec	Passive	same as above
Pa_T_T_SynthArray	Passive	same as above
Pa_T_T_Scale	Passive	same as above
Pa_T_T_ReductionMath	Passive	same as above
Pa_T_T_Logarithm	Passive	same as above
Pa_T_T_Power	Passive	same as above
Pa_T_T_Unary	Passive	same as above
Pa_T_T_Trigonometry	Passive	same as above
Pa_T_T_Binary	Passive	same as above

Graphical Description

FIGURE 23. Pa_T_Transform Object Hood Graphical Description



21.0 Pa_T_Present

21.1 Problem Definition

The **Present** module contains a large number of modules that display the data in various graphical ways. Each individual method is provided by a submodule in the lower level and is referred to as a presentation functional unit. The purpose of the various presentation modules is clear. Each one serves as different intuitive way of presenting the analyzed data.

21.2 Formalization of the strategy

The various presentation functional units are implemented as objects with almost similar operations. As mentioned earlier the presentation functional units are usually termi-

nal nodes in the analysis graph. They accept an input connection and present the data in various ways. Some of them require that the data have been previously transformed in a higher dimension such as vector or array.

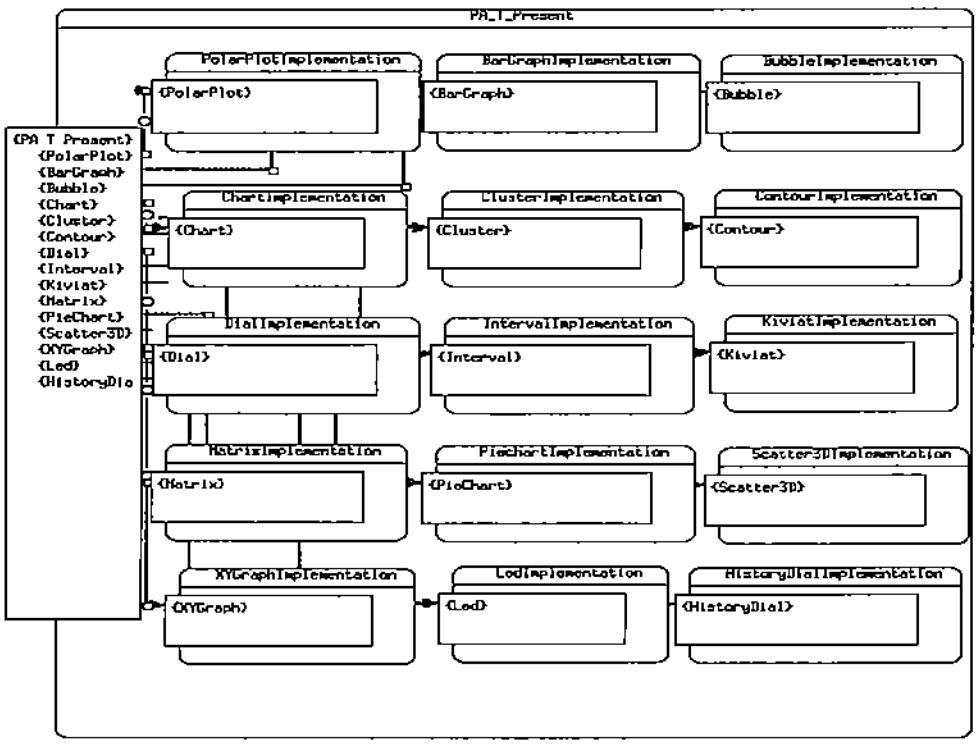
Identification of the Objects and Operations

TABLE 33. Pa_T_Present Objects and Operations

Object Name	Object Type	Object Operations
Pa_T_P_Polarplot	Passive	Pa_T_T_PP_Run Pa_T_T_PP_Ready Pa_T_T_PP_SaveToFile Pa_T_T_PP_LoadFromFile Pa_T_T_PP_Init Pa_T_T_PP_Copy Pa_T_T_PP_Configure
Pa_T_P_Bargraph	Passive	same as above
Pa_T_P_Bubble	Passive	same as above
Pa_T_P_Chart	Passive	same as above
Pa_T_P_Cluster	Passive	same as above
Pa_T_P_Contour	Passive	same as above
Pa_T_P_Dial	Passive	same as above
Pa_T_P_Interval	Passive	same as above
Pa_T_P_Kiviat	Passive	same as above
Pa_T_P_Matrix	Passive	same as above
Pa_T_P_Piechart	Passive	same as above
Pa_T_P_Scatter3D	Passive	same as above
Pa_T_P_XYGraph	Passive	same as above
Pa_T_P_Led	Passive	same as above
Pa_T_P_HistoryDial	Passive	same as above

Graphical Description

FIGURE 24. Pa_T_Present Object Hood Graphical Description



22.0 Pa_T_Display

22.1 Problem Definition

The Display module offers a number of advanced data display techniques which can be viewed as aggregate data displays. The reason that these techniques are grouped differently than the presentation techniques is mainly conceptual. The presentation functions are in a sense generic, that is, they can be applied to a number of different data streams. The kind of displays we implement in the display module however are more custom designed in the sense that they assume some properties of the data we are analyzing.

22.2 Formalization of the strategy

The aggregate display functionality is grouped into two functional units, the animation and the zooming. Both units are implemented as objects which provide a similar set of operations, conforming with the specification of the rest of the functional units.

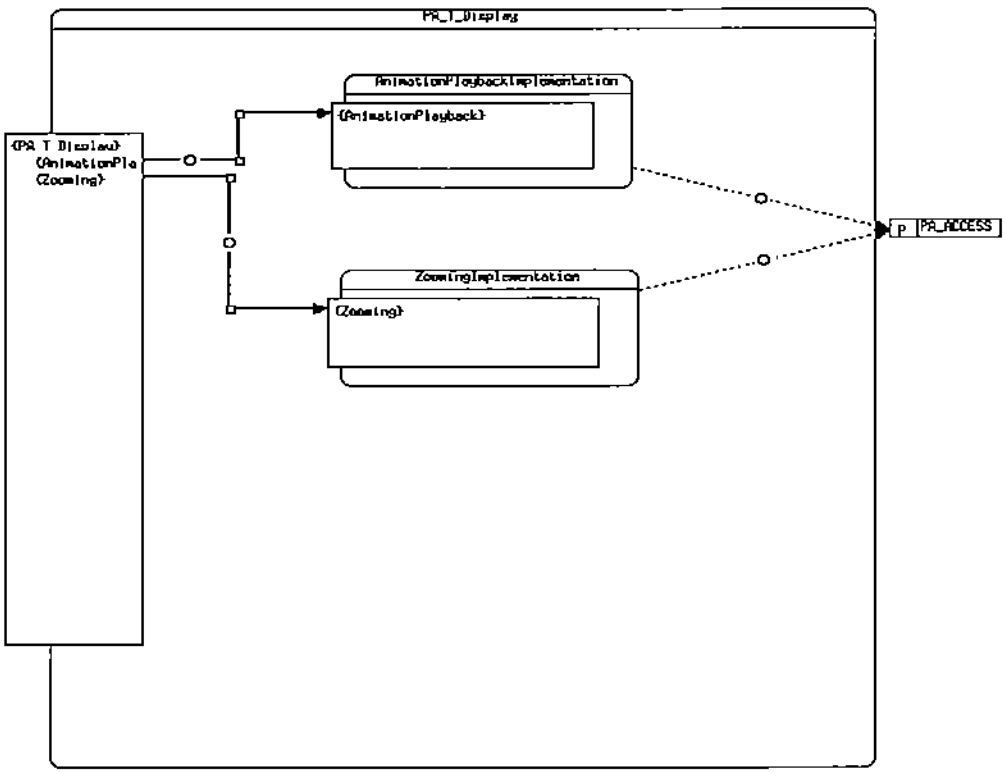
Identification of the Objects and Operations

TABLE 34. Pa_T_Display Objects and Operations

Object Name	Object Type	Object Operations
Pa_T_D_Animation	Passive	Pa_T_D_A_SavetoFile
		Pa_T_D_A_LoadfromFile
		Pa_T_D_A_Speed
		Pa_T_D_A_Type
		Pa_T_D_A_Ready
		Pa_T_D_A_Run
		Pa_T_D_A_Init
		Pa_T_D_A_Configure
Pa_T_D_Zooming	Passive	Pa_T_D_Z_Restart
		Pa_T_D_Z_LoadfromFile
		Pa_T_D_Z_SavetoFile
		Pa_T_D_Z_Callbacks
		Pa_T_D_Z_Ready
		Pa_T_D_Z_Init
		Pa_T_D_Z_Configure
		Pa_T_D_Z_Run

Graphical Description

FIGURE 25. Pa_T_Display Object Hood Graphical Description



23.0 Pa_Access

23.1 Problem Definition

The Access module encloses all the functionality needed to deal with accessing the data stored by the monitoring system. It provides a set of functions necessary to convert among trace formats, a set of functions needed to perform queries to obtain trace data, and an optional set of database operations. The main stream functions are those performing the query support on the data. The converters are used so as to provide portability and universality of the Patool in terms of trace data accepted.

23.2 Formalization of the strategy

The functionality provided by the Access module is separated in three different submodules. The Database operations are currently not supported, but its provision is con-

sidered important and for that reason its interface is identified. The other two modules are the Convertion and the Query.

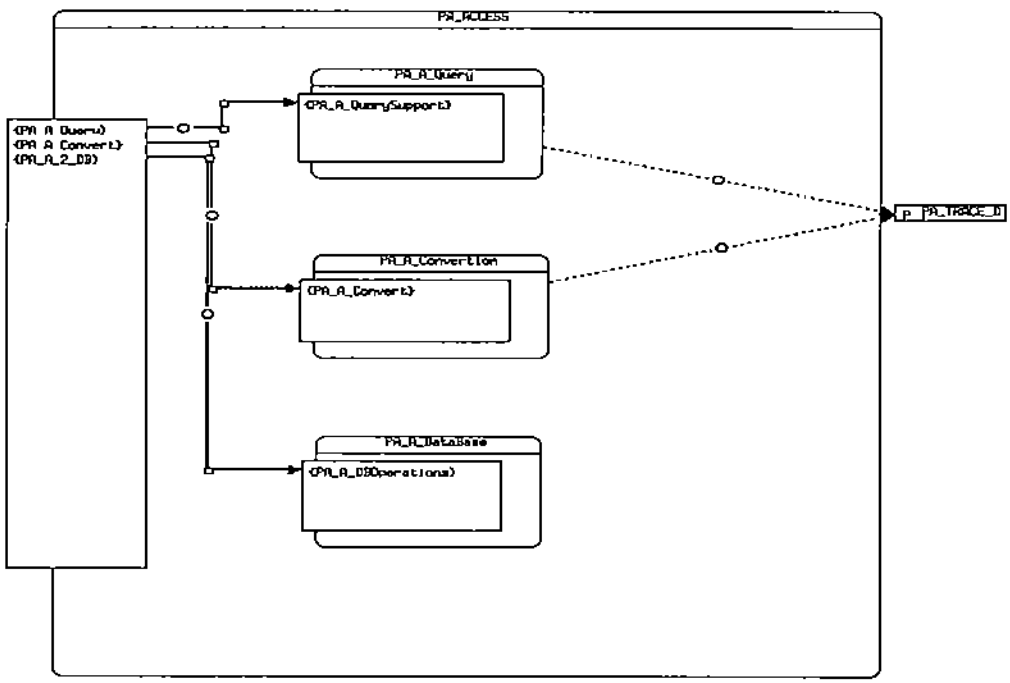
Identification of the Objects and Operations

TABLE 35. Pa_Access Objects and Operations

Object Name	Object Type	Object Operations
Pa_A_Query	Passive	Pa_A_Q_OpStruct Pa_A_Q_OpField Pa_A_Q_OpTag Pa_A_Q_Op_Attr
Pa_A_Convertion	Passive	Pa_A_C_tosddf Pa_A_C_fromsddf

Graphical Description

FIGURE 26. Pa_Access Object Hood Graphical Description



24.0 Pa_A_Query

24.1 Problem Definition

The **Query** module encloses all the functionality needed to access the stored trace data. It is implemented in a way so as to be able to handle the Sddf trace format in a flexible manner. The functionality is provided via a number of different modules that implement the strategy we use to access the trace data.

24.2 Formalization of the strategy

The Query module consists of a number of objects that deliver all the necessary functions to manage the trace data.

A dictionary is a set of names of some entities for which a definition has been discovered. It is used heavily throughout the system for the management of the information collected. A descriptor is an encapsulation of the knowledge we have with respect to a specific object. An Iterator is an encapsulation offering a number of operations for managing iterative data structures such as lists. Its services are used heavily by the system since most of the data handling is done using lists. A Pipe is similar in notion with an ordinary pipe of the Unix system,. However, here the functionality is customized to be used with the functional units and the general framework of the data management. For example when two functional units are connected in the DAG, in the low level this is achieved by a pipe connecting them.

Identification of the Objects and Operations

TABLE 36. Pa_A_Query Objects and Operations

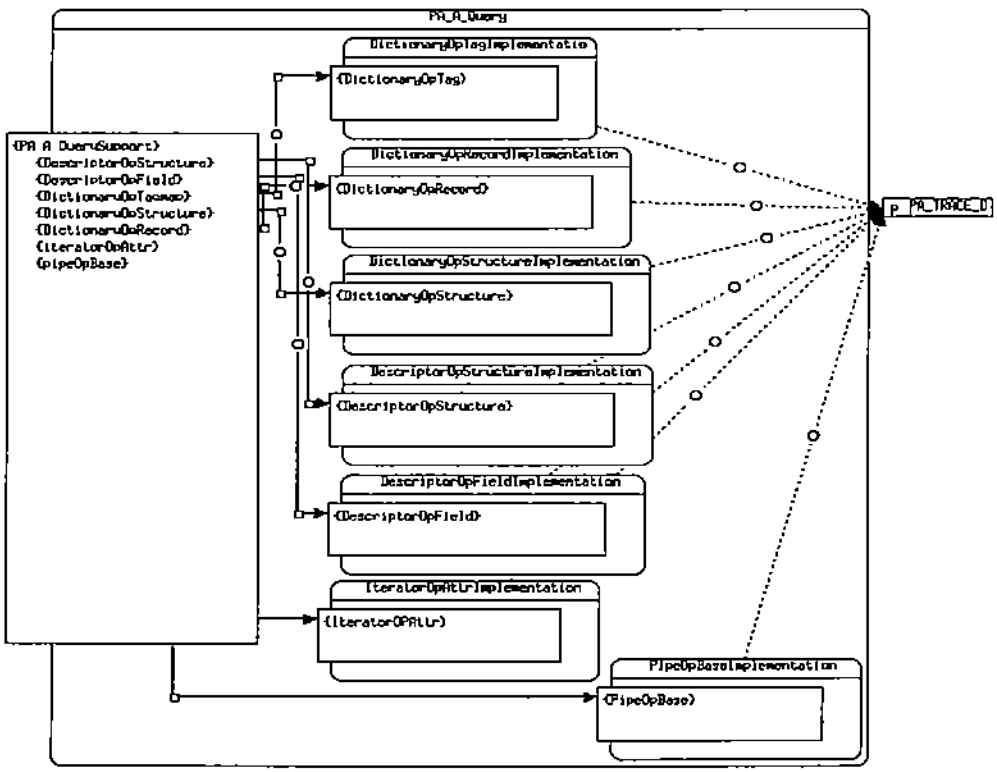
Object Name	Object Type	Object Operations
Pa_A_Q_PipeOpBase	Passive	Pa_A_Q_P_setOutWrapper Pa_A_Q_P_setInWrapper Pa_A_Q_P_getOutWrapper Pa_A_Q_P_getInWrapper Pa_A_Q_P_getObjName
Pa_A_Q_IteratorOpAttr	Passive	Pa_A_Q_I_AttrCreate Pa_A_Q_I_AttrNext Pa_A_Q_I_AttrValue Pa_A_Q_I_AttrFirst Pa_A_Q_I_AttrDestroy
Pa_A_Q_DescriptorOpStruct	Passive	Pa_A_Q_DeS_StructRemove Pa_A_Q_DeS_StructInsert Pa_A_Q_DeS_StructFetch Pa_A_Q_DeS_entryCount Pa_A_Q_DeS_StructContains Pa_A_Q_DeS_bytesNeeded Pa_A_Q_DeS_bitsToObj
Pa_A_Q_DescriptorOpField	Passive	Pa_A_Q_DeF_SetDimension Pa_A_Q_DeF_SetType Pa_A_Q_DeF_GetType Pa_A_Q_DeF_GetDimension Pa_A_Q_DeF_bytesNeeded Pa_A_Q_DeF_bitsToObj
Pa_A_Q_DictionaryOpRec	Passive	Pa_A_Q_DiR_RecInsert Pa_A_Q_DiR_RecGetValue Pa_A_Q_DiR_RecGetArray Pa_A_Q_DiR_RecFetch Pa_A_Q_DiR_entryCount Pa_A_Q_DiR_RecContains Pa_A_Q_DiR_RecSetValue

TABLE 36. Pa_A_Query Objects and Operations

Object Name	Object Type	Object Operations
Pa_A_Q_DictionaryOpTag	Passive	Pa_A_Q_DiT_TagSavetoFile Pa_A_Q_DiT_TagLoadfromFile Pa_A_Q_DiT_TagInsert Pa_A_Q_DiT_TagFetch Pa_A_Q_DiT_TagClearAll
Pa_A_Q_DictionaryOpStruct	Passive	Pa_A_Q_DiS_StructToFle Pa_A_Q_DiS_StructInsert Pa_A_Q_Dis_StructFetch

Graphical Description

FIGURE 27. Pa_A_Query Object Hood Graphical Description



25.0 Pa_A_Convertion

25.1 Problem Definition

The **Convertion** module encloses all the functionality that is needed to deal with the various demands for interchanging through data formats, importing and exporting data specifications and providing the possibility for a universal many to many mapping of the trace data assuming semantic compatibility. It provides a set of convertion functions among the various sources and targets. The services are implemented by different modules which are enclosed.

25.2 Formalization of the strategy

The Convertion module is an open ended collection of convertion routines that pertain to the format of trace data. The bottom line is common and it is the Sddf format, either in binary or in ascii.

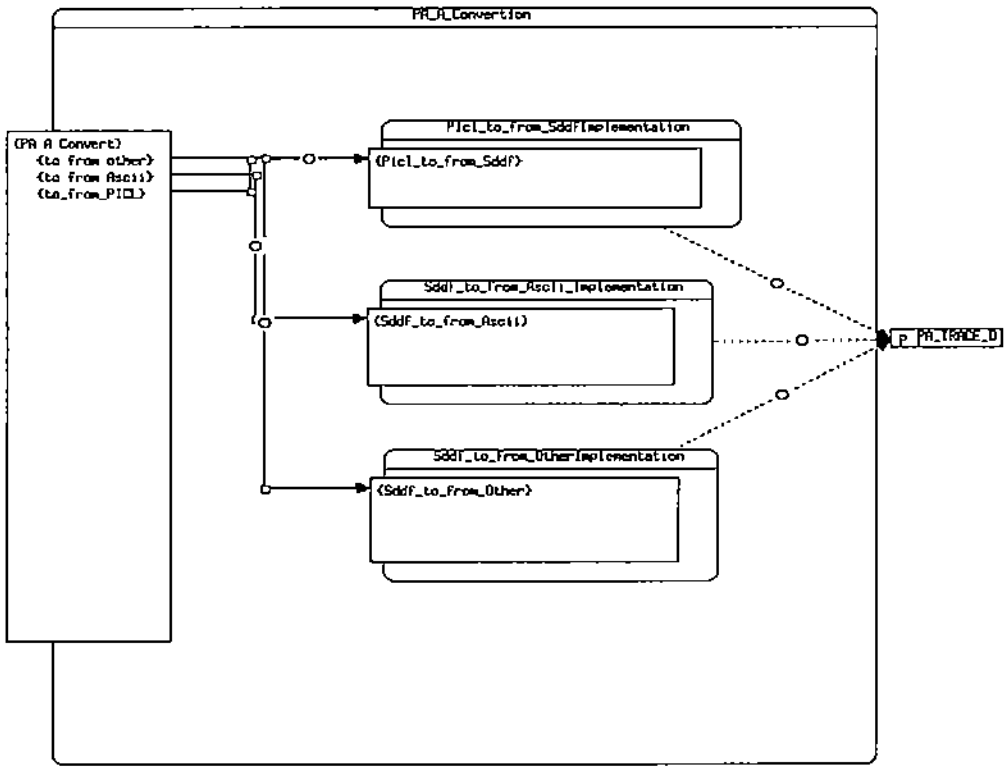
Identification of the Objects and Operations

TABLE 37. Pa_A_Convertion Objects and Operations

Object Name	Object Type	Object Operations
Pa_A_C_PiclToFromSDDF	Passive	Pa_A_C_Sddf2Picl Pa_A_C_Picl2Sddf
Pa_A_C_SddfToFromAscii	Passive	Pa_A_C_Sddf2Ascii Pa_A_C_Ascii2Sddf
Pa_A_C_SddfToFromOther	Passive	Pa_A_C_Sddf2Other Pa_A_C_Other2Sddf

Graphical Description

FIGURE 28. Pa_A_Convertion Object Hood Graphical Description



26.0 Pa_A_TraceData

26.1 Problem Definition

The Trace Data module encapsulates the definition of the various trace formats that might be understood and that are accepted by the Patool. It provides a set of definitions to the converters and the other functions that deal with the trace data. It is implemented as a separate module for modularity and extensibility.

26.2 Formalization of the strategy

The notion of trace data is separated in those traces that are in a native Sddf format and those which are originally stored in some other format and for which we have available an Sddf structural description and a corresponding converter.

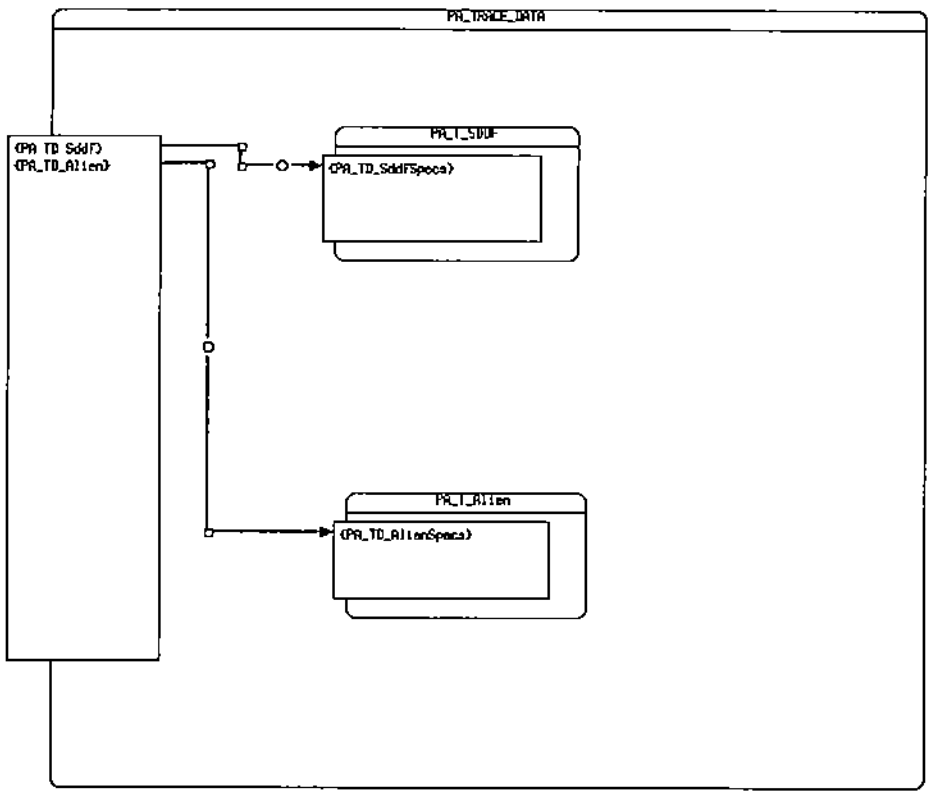
Identification of the Objects and Operations

TABLE 38. Pa_A_TraceData Objects and Operations

Object Name	Object Type	Object Operations
Pa_T_Sddf	Passive	Pa_T_Sddf
Pa_T_Alien	Passive	Pa_T_Alien

Graphical Description

FIGURE 29. Pa_A_TraceData Object Hood Graphical Description



27.0 Formal Documentation Cross References

27.1 HOOD Objects and Source Code

This section gives a quantitative description of the HOOD objects comprising the Patool. Direct references to source code files implementing the various objects are given.

1. **Pa_User_Interface:**

- *Pa_U_Control:*

Pa_U_C_MainControl:
FUConfig/ParamDisplay.C
Includes/ParamDisplay.h
Interface/GraphDisplay.C
Includes/GraphDisplay.h
Interface/Patool.C
Includes/Patool.h
Interface/PatoolMainInterface.C
Includes/PatoolMainInterface.h

Pa_U_C_menus:
Includes/PatoolMainMenus.def
Interface/PatoolMainInterface.C
Includes/PatoolMainInterface.h

Pa_U_C_tool_resources:
./AddModuleDialog
./AnimationFU
/BargraphFU
./BoxAndWhisker
./BubbleFU
./ChartFU
./ClusterFU
./ColorEditFU
./ConfigBoardDetail
./ConstantsDialog
./ContourFU
./DefaultsDialog
./DelayDialog
./DialFU
./ExecutionModeDialog
./FUDialog
./FUParamConfig
./FUParamDisplay
./FileInputDialog
./FileMergeDialog
./GanttFU
./GeneralDefaults
./GraphFU
./HelpSystem
/HistoryDialFU
./IntervalFU
./KiviatFU
./LedFU
./MatrixFU
./MultivariateFU
./OutputSpecificationDialog

./ParallelCoords
./PiechartFU
./PipePortBindingDialog
./PolarPlotFU
./PrinterFUDialog
./ProfileFU
./RecordBindingDialog
./RunDialog
./Scatter3dFU
./ScatterPlotMatrix
./Snapshot
./SocketInputDialog
./StatusBoard
./TranscriptDialog
./UtilFU
./XYGraph
./XYGraphFU

- *Pa_U_Display*:

DefaultsDialogImplementation:
Interface/DefaultsDialog.C
Includes/DefaultsDialog.h

FileSelectionDialogImplementation:
Interface/FileSelectionDialog.C
Includes/FileSelectionDialog.h

GeneralDialogBoxImplementation:
Interface/GeneralDialogBox.C
Includes/GeneralDialogBox.h
Interface/AddModuleDialog.C
Includes/AddModuleDialog.h
Interface/ConfigBoard.C
Includes/ConfigBoard.h
Interface/ConfigBoardDetail.C
Includes/ConfigBoardDetail.h
Interface/ConstantsDialog.C
Includes/ConstantsDialog.h
Interface/DelayDialog.C
Includes/DelayDialog.h
Interface/ExecutionModeDialog.C
Includes/ExecutionModeDialog.h
Interface/FUClassification.C
Includes/FUClassification.def
Includes/FUClassification.h
Interface/OutputSpecificationDialog.C
Includes/OutputSpecificationDialog.h
Interface/PipePortBindingDialog.C
Includes/PipePortBindingDialog.h
Interface/RecordBindingDialog.C
Includes/RecordBindingDialog.h
Interface/RunDialog.C
Includes/RunDialog.h
Interface/SnapshotDialog.C
Includes/SnapshotDialog.h
Interface/TranscriptWindow.C
Includes/TranscriptWindow.h

- *Pa_U_Tools*

ConnectOtherImplementation
Has not been implemented yet

.

- *Pa_U_Help*:

HelpImplementation
XInterface/Help.C
Includes/Help.h
XInterface/PatoolHelpSystem.C
Includes/PatoolHelpSystem.h

2. **Pa_Manager**:

- *Pa_M_Data*:

DataImplementation:
Wrapper/FUWrapper.C
Includes/FUWrapper.h
Wrapper/FileInputWrapper.C
Includes/FileInputWrapper.h
Wrapper/FileMergeWrapper.C
Includes/FileMergeWrapper.h
Wrapper/FileOutputWrapper.C
Includes/FileOutputWrapper.h
Wrapper/InterfaceClass.C
Includes/InterfaceClass.h
Wrapper/SocketInputDisplay.C
Includes/SocketInputDisplay.h
Wrapper/SocketInputWrapper.C
Includes/SocketInputWrapper.h
Wrapper/Wrapper.C
Includes/Wrapper.h
Streams/SddfProcessor.C
Includes/SddfProcessor.h
Streams/Sddfattributes.C
Includes/Sddfattributes.h
Streams/Sddfcommand.C
Includes/Sddfcommand.h
Streams/SddFdata.C
Includes/SddFdata.h
Streams/SddFdescriptor.C
Includes/SddFdescriptor.h
Streams/Sddfheader.C
Includes/Sddfheader.h
Streams/Sddfmerge.C
Includes/Sddfmerge.h
Streams/Sddfstrbase.C
Includes/Sddfstrbase.h

- *Pa_M_RunTime*:

RunTSImplementation:
Interface/PatoolMainInterface.C
Includes/PatoolMainInterface.h

- *Pa_M_DisPatch*:

DisPatchingImplementation:
Wrapper/ActionRecord.C
Includes/ActionRecord.h

Wrapper/ActionTable.C
Includes/ActionTable.h
Wrapper/InputPipeSocket.C
Includes/InputPipeSocket.h
Wrapper/InputPort.C Includes/InputPort.h
Wrapper/OutputPort.C
Includes/OutputPort.h
Managers/ConnectionId.C
Includes/ConnectionId.h
Managers/ConnectionIdPtrList.C
Includes/ConnectionIdPtrList.h
Managers/FunctionalUnitManager.C
Includes/FunctionalUnitManager.h
Managers/InfrastructureManager.C
Includes/InfrastructureManager.h
Managers/ModuleId.C
Includes/ModuleId.h
Managers/ModuleIdPtrList.C
Includes/ModuleIdPtrList.h

3. Pa_Toolkit:

- *Pa_T_Transform:*

ScaleImplementation:
FunctionalUnits/ScaleFU.C
Includes/ScaleFU.h

ReductionMathImplementation:
FunctionalUnits/ReductionMathFU.C
Includes/ReductionMathFU.h

LogarithmImplementation:
FunctionalUnits/LogarithmFU.C
Includes/LogarithmFU.h

PowerImplementation:
FunctionalUnits/PowerFU.C
Includes/PowerFU.h

UnaryImplementation:
FunctionalUnits/UnaryMathFU.C
Includes/UnaryMathFU.h

BinaryImplementation:
FunctionalUnits/BinaryMathFU.C
Includes/BinaryMathFU.h

TrigonometryImplementation:
FunctionalUnits/TrigonometryFU.C
Includes/TrigonometryFU.h

SynthCoordImplementation:
FunctionalUnits/SynthesizeCoordinates.C
Includes/SynthesizeCoordinates.h

SynthVecImplementation:
FunctionalUnits/SynthesizeVector.C
Includes/SynthesizeVector.h
FunctionalUnits/SynthesizeVectorElement.C
Includes/SynthesizeVectorElement.h

SynthArrayImplementation:

FunctionalUnits/SynthesizeArray.C
Includes/SynthesizeArray.h
FunctionalUnits/SynthesizeArrayElemenLC
Includes/SynthesizeArrayElement.h

- *Pa_T_Present*:

PolarPlotImplementation:
FunctionalUnits/PolarPlotFU.C
Includes/PolarPlorFU.h

BarGraphImplementation:
FunctionalUnits/BarGraphFU.C
Includes/BarGraphFU.h

BubbleImplementation:
FunctionalUnits/BubbleFU.C
Includes/BubbleFU.h

ChartImplementation:
FunctionalUnits/ChartFU.C
Includes/ChartFU.h

ClusterImplementation:
FunctionalUnits/ClusterFU.C
Includes/ClusterFU.h

ContourImplementation:
FunctionalUnits/ContourFU.C
Includes/ContourFU.h

DialImplementation:
FunctionalUnits/DialFU.C
Includes/DialFU.h

IntervalImplementation:
FunctionalUnits/IntervalFU.C
Includes/IntervalFU.h

KiviatImplementation:
FunctionalUnits/KiviatFU.C
Includes/KiviatFU.h
FunctionalUnits/KivialUtilDisplay.C
Includes/KiviatUtilDisplay.h

MatrixImplementation:
FunctionalUnits/MatrixFU.C
Includes/MatricFU.h
FunctionalUnits/ScatterPlotMatrixFU.C
Includes/ScatterPlotMatrixFU.h

PiechartImplementation:
FunctionalUnits/PiechartFU.C
Includes/PiechartFU.h

Scatter3DImplementation:
FunctionalUnits/Scatter3DFU.C
Includes/Scatter3DFU.h

LedImplementation:
FunctionalUnits/LedFU.C
Includes/LedFU.h

HistoryDialImplementation:
FunctionalUnits/HistoryDialFU.C
Includes/HistoryDialFU.h

- *Pa_T_Display*:

AnimationPlaybackImplementation:
FunctionalUnits/AnimationFU.C
Includes/AnimationFU.h

- *Pa_T_Extent*:

4. **Pa_FdSuit**:

- *Pa_Fd_FilterReduce*:
FunctionalUnits/FiAndReFU.C
Includes/FiAndReFU.h
FunctionalUnits/FilterRecordsFU.C
Includes/FilterRecordsFU.h
- *Pa_Fd_GEditor*:

GraphEdgeOpsImplementation:
Interface/GraphEdge.C
Includes/GraphEdge.h
Interface/GraphEdgePtrList.C
Includes/GraphEdgePtrList.h

GraphNodeOpsImplementation:
Interface/GraphNode.C
Includes/GraphNode.h
Interface/GraphNodePtrList.C
Includes/GraphNodePtrList.h

GraphDisplayOpsImplementation:
Interface/GraphDisplay.C
Includes/GraphDisplay.h

5. **Pa_Access**:

- *Pa_A_Query*:

DictionaryOpTagImplementation:
Dictionaries/TagMappingDictionary.C
Includes/TagMappingDictionary.h

DictionaryOpRecordImplementation:
Dictionaries/RecordDictionary.C
Includes/RecordDictionary.h

DictionaryOpStructureImplementation:
Dictionaries/StructureDictionary.C
Includes/StructureDictionary.h

DescriptorOpStructureImplementation:
Descriptors/StructureDescriptor.C
Includes/StructureDescriptor.h
Dictionaries/StructureDescriptorResolver.C
Includes/StructureDescriptorResolver.h

DescriptorOpFieldImplementation:
Descriptors/FieldDescriptor.C
Includes/FieldDescriptor.h

IteratorOPattrImplementation:
Iterators/AttributesIterator.C
Includes/AttributesIterator.h

Iterators/RecordDictionaryIterator.C
Includes/RecordDictionaryIterator.h
Iterators/RecordDictionaryPIterator.C
Includes/RecordDictionaryPIterator.h
Iterators/StructureDescriptorIterator.C
Includes/StructureDescriptorIterator.h
Iterators/StructureDictionaryInquirer.C
Includes/StructureDictionaryInquirer.h
Iterators/StructureDictionaryIterator.C
Includes/StructureDictionaryIterator.h

PipeOpBaseImplementation:

Pipes/AsciiPipeReader.C
Includes/AsciiPipeReader.h
Pipes/AsciiPipeWriter.C
Includes/AsciiPipeWriter.h
Pipes/BasePipe.C
Includes/BasePipe.h
Pipes/BinaryPipeReader.C
Includes/BinaryPipeReader.h
Pipes/BinaryPipeWriter.C
Includes/BinaryPipeWriter.h
Pipes/ConversionPipeReader.C
Includes/ConversionPipeReader.h
Pipes/ConversionPipeWriter.C
Includes/ConversionPipeWriter.h
Pipes/FIFOStreamPipe.C
Includes/FIFOStreamPipe.h
Pipes/FileStreamPipe.C
Includes/FileStreamPipe.h
Pipes/InputFileStreamPipe.C
Includes/InputFileStreamPipe.h
Pipes/InputSocketStreamPipe.C
Includes/InputSocketStreamPipe.h
Pipes/MergePipeReader.C
Includes/MergePipeReader.h
Pipes/MergeStreamPipe.C
Includes/MergeStreamPipe.h
Pipes/OutputFileStreamPipe.C
Includes/OutputFileStreamPipe.h
Pipes/PipeReader.C
Includes/PipeReader.h
Pipes/PipeWriter.C
Includes/PipeWriter.h
Pipes/SocketStreamPipe.C
Includes/SocketStreamPipe.h
Pipes/StreamPipe.C
Includes/StreamPipe.h